NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

DESIGN OF A
SIXTEEN BIT PIPELINED ADDER
USING CMOS BULK P-WELL TECHNOLOGY

by

William R. Reid

December 1984

Thesis Advisor: D. E. Kirk

Approfed for public release; distribution unlimited

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)*: Design of a Sixteen Bit Pipelined Adder Using CMOS Bulk P-Well Technology	5. TYPE OF REPORT & PERIOD COVERED: Master's Thesis; December 1984
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): William R. Reid	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School, Monterey, California 93943	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Postgraduate School, Monterey, California 93943	12. REPORT DATE: December 1984
	13. NUMBER OF PAGES: 116
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*	15. SECURITY CLASS. *(of this report)*: UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

VLSI Design, CMOS, CMOS-PW, Pipelined Adder, Carry Look Ahead Addition, CAD Tools

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

The design of a sixteen-bit pipelined adder CMOS integrated circuit is presented. The adder is designed to maximize throughput and to provide for testability. Tutorial material on CMOS design is also presented.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

Approved for public release; distribution is unlimited.

Design of a Sixteen Bit Pipelined Adder Using CMOS Bulk P-Well Technology

by

William R. Reid
Lieutenant Commander, United States Navy
B.S., Purdue University, 1975

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December 1984

ABSTRACT

The design of a sixteen-bit pipelined adder CMOS integrated circuit is presented. The adder is designed to maximize throughput and to provide for testability. Tutorial material on CMOS design is also presented.

TABLE OF CCNTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

For several years the ability of systems engineers to design custom digital integrated circuits has been growing. The Mead and Conway design methodology described in Introduction to VLSI Systems [Ref. 1], permits the systems engineer to be his own logic circuit designer. A proliferation of computer-aided design (CAD) systems such as the MacPitts silicon compiler [Ref. 2], the chip layout language (CLL) [Ref. 3], the graphics editor Caesar [Ref. 4], and the Burlap hierarchical layout language [Ref. 5] make it possible for the engineer to rapidly carry the Mead and Conway design methodology through to a final design. This includes iterative simulation and redesign to provide justifiable confidence in the final design submitted for fabrication.

Many of the techniques utilized in the Mead and Conway methodology and most of the CAD tools are based on having the final design implemented in a technology that uses only one type of doping for the semiconductor material in the active region of the transistors. Because of their higher switching speed, negatively doped metal oxide semiconductor (NMOS) transistor technologies are generally used.

Selection of an NMOS implementation technology does provide the systems engineer with a complete and proven methodology for the design of a very large scale integrated (VLSI) circuit and allows the use of many extensively tested CAD tools. Like any other design decision, selection of NMOS implementation brings with it some limitations. There are two primary problems associated with NMOS digital circuits.

The first is the ultimate switching speed limitation. Though many NMOS VLSI circuits operate at clock rates in the 8 to 10 MHz range, there are many applications requiring higher clock rates. The second problem is the dissipation of the relatively large amount of power consumed by NMOS digital circuits. State of the art, commercially available NMOS VLSI circuits commonly have power consumptions in the vicinity of 3 to 5 watts. Considerable design effort is required to insure that the dissipation of this much energy by a chip measuring approximately 5 millimeters on a side does not alter the performance of the micron sized features on the chip.

One group of technologies that offers both increased switching speed and greatly reduced power consumption is complementary metal oxide semiconductors (CMOS). CMOS circuits also offer the benefits of greater radiation hardening and increased noise margin. In this thesis investigation, much of the Mead and Conway methodology was utilized in the design of a CMOS circuit. A general purpose color graphics CAD tool called Caesar that has been frequently used in the design of NMOS circuits was employed. In carrying out the design of the 16 bit pipelined high speed adder in CMOS two separate goals were pursued. The first, of course, is speed and the second is verifiability. A high speed adder implies not only a high clock rate of operation but also a small latency between input of operands and output of the sum.

A discussion of CMOS technologies and the implementation of logic circuits in those technologies follows in Chapter 2. Chapter 3 presents a description of the CAD tools used to construct and simulate the layout for the adder. The logic and layout design of the adder is covered in Chapter 4 and is followed by a test plan for the fabricated chip in Chapter 5.

II. CMOS CIRCUITS

Before the design of CMOS digital circuits can be attempted, an understanding of how to best implement logic functions in CMOS is necessary. It is also important to be aware of the advantages and disadvantages of the different CMOS implementation technologies. In this chapter the operation of CMOS digital circuits is explained using similar NMOS circuits as a benchmark for comparison. The different methodologies for assembling the CMOS pieces to produce the desired logical results are reviewed and the selection of the CMOS-Bulk p-well implementation technology is explained.

A. COMPARISON WITH NMOS

In NMOS digital circuits there is only one type of switching device, namely the n-channel enhancement mode metal oxide semiconductor (MOS) transistor. The other principal device utilized in NMOS circuits is the depletion mode n-channel MOS device which acts as a load resistor. In CMOS there are both n-channel and p-channel enhancement mode transistors available. As in NMOS, the n-channel device can be considered on when Vdd (typically +5 Volts DC), a logical 1, is present on its gate. The p-channel device can be considered on when ground (GND), a logical 0, is present on its gate. In Figure 2.1 are the symbols that will be used for the n-channel and p-channel transistors in this thesis.

The basic differences between NMOS and CMOS technologies can be demonstrated by comparing their application to some basic digital circuits.

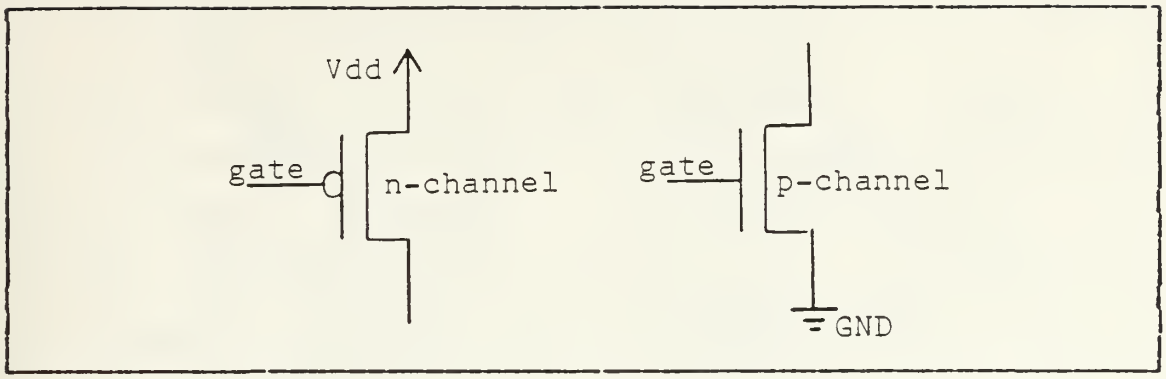


Figure 2.1 CMOS Transistor Symbols.

1. The Inverter

Figure 2.2 (a) shows an NMOS inverter. Whenever there is a logical 1 on the input, the voltage drop across the load resistor is approximately Vdd and the output is a logical 0. This results in steady state power consumption. When the input switches to a logical 0, before the output can assume a logical 1, the load capacitance (C1) on the output must be charged to Vdd through the load resistor with a resistance of several kilohms. This results in a much longer transition from 0 to 1, where the load capacitance is charged through the load resistor, than from 1 to 0 where the load capacitance is discharged through the switched on NMOS enhancement transistor. The reason for this asymmetry is that the pull-down transistor's on resistance is typically only one fourth or less that of the on resistance of the pull-up load depletion mode transistor. The technique of precharging circuits, where all outputs are set to logical 1 during one clock cycle and then selectively forced to 0 on the opposite (evaluation) clock cycle has proven helpful in gaining control over the unsymmetric switching times. This longer switching time from 0 to 1 must still be accounted for, however, and represents the primary limitation to the speed of NMOS circuits.

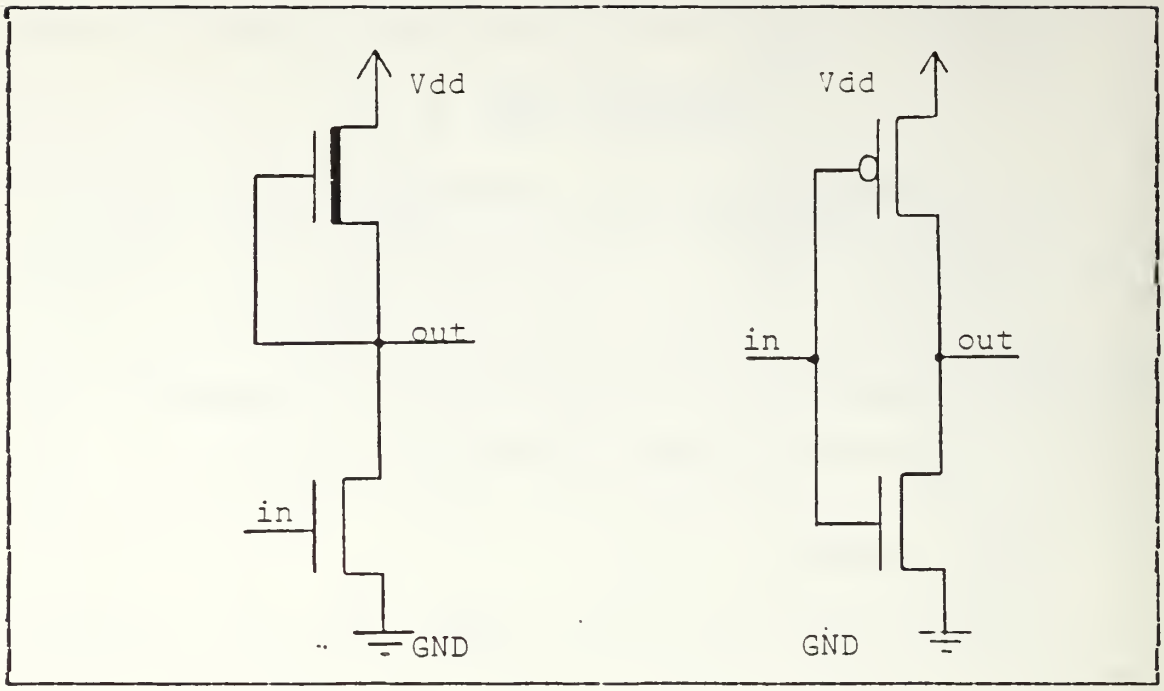


Figure 2.2 (a) NMOS Inverter (b) CMOS Inverter.

In the CMOS inverter of Figure 2.2 (b) the input is applied to the gates of both devices. An input of logical 1 causes the n-channel device to switch on and the p-channel device to switch off, resulting in an output of logical 0. Similarly, an input of 0 results in an output of 1. In both cases, one device is fully off, representing a resistance on the order of gigaohms. Thus, the steady state power consumption is essentially zero. In operation the only power consumption of consequence occurs during the transition when neither transistor is fully on or off. Additionally, since the output load capacitance is both charged and discharged through a turned on transistor, the 1 to 0 and 0 to 1 switching delays are theoretically the same.

Actually the switching delays depend on many parameters. The n-channel and p-channel device dimensions are frequently not the same, the mobility of the electrons in

the n-channel is greater than the mobility of the holes in the p-channel. Also, the capacitive load seen by the p-channel device in CMOS p-well (CMOS-pw) is greater than the load seen by the n-channel device because of the highly doped p-well. Typically, the result in CMOS-pw is a slightly longer transition time of the 0 to 1 output transition. Some designers attempt to compensate for this by consistently making the p-channel transistors wider than the n-channel transistors.

Unlike NMOS, the output of a CMOS digital circuit makes a full excursion between Vdd and GND. This makes CMOS circuits less sensitive to noise than NMOS circuits. CMOS should also benefit more from future reductions in feature size. NMOS is more restricted in ultimate feature size because the power dissipation requirements of the depletion mode devices will create more problems as feature sizes shrink. In Figure 2.3 the relative sizes of minimum dimension inverters implemented in currently available 3 micron feature size CMOS-PW and NMOS technologies are shown.

2. The NOR Gate and Transmission Gate

Figure 2.4 shows the circuit diagrams and layouts of a two-input NOR gate implemented in both CMOS-PW and NMOS. From Figures 2.3 and 2.4 it is evident that static[1] CMOS gates are more complex and area consuming than their NMOS counterparts. In these fully complementary circuits a redundancy in the structures is evident. The pull-up only or pull-down only would be sufficient to implement the logic. In the CMOS circuits of Figures 2.3 and 2.4 the inputs must perform two tasks. A logical 1 on an input causes both a connection between the output and ground and a

---

[1]Static logic circuits continuously evaluate their inputs and produce their specified logic output. Dynamic circuits perform logical evaluation of the inputs only when directed to do so by control signals and/or clock signals.

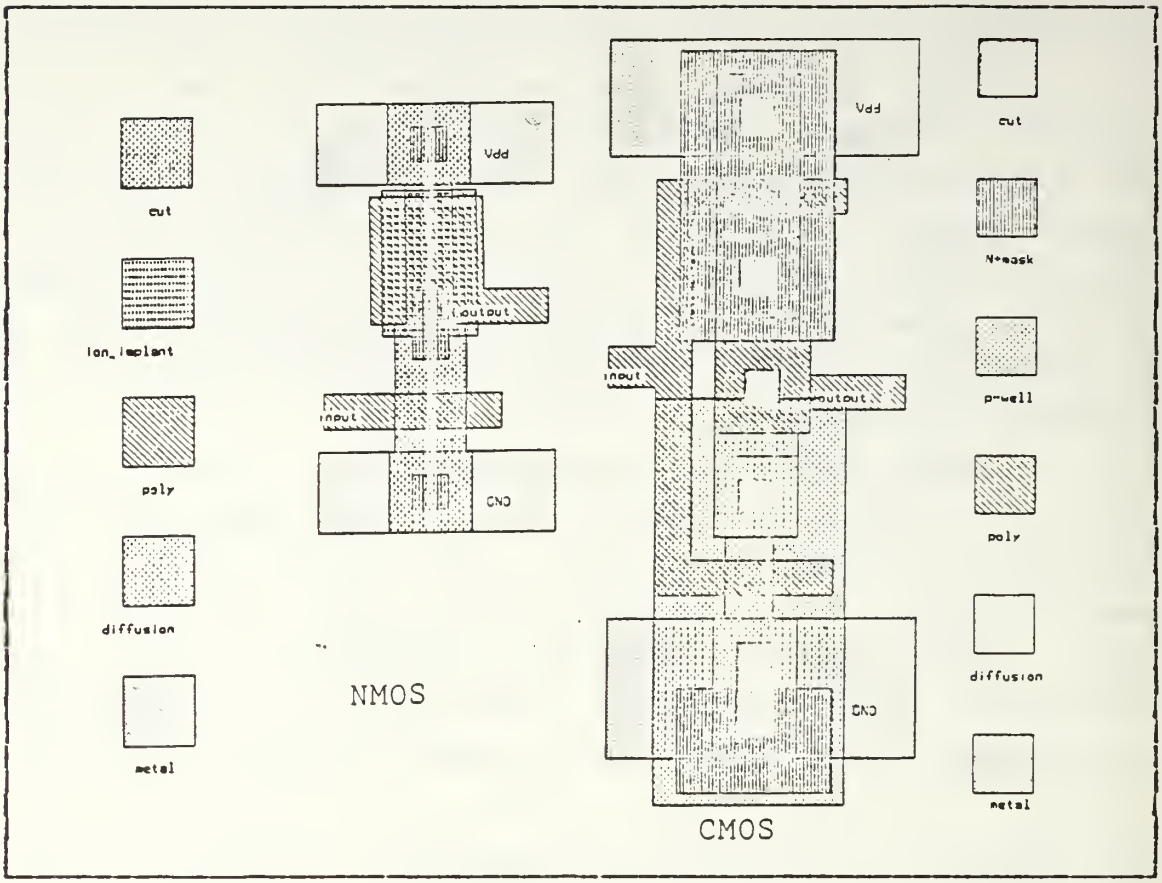


Figure 2.3 Minimum Dimension Inverters.

disconnection between the output and Vdd. Logically these two actions are equivalent, therefore only one action should be necessary to implement the logic. Design methodologies to accomplish this are described in section B of this chapter. The parallelism of the CMOS transmission gate of Figure 2.5 and the NMOS pass transistor is evident. The major difference lies in the bilateral nature of the CMOS transmission gate. It is made up of both n-channel and p-channel devices and requires both polarities of the control signal for operation. The reason for this bilateral requirement is that the p-channel device does not transmit low voltages well and the n-channel device does not transmit

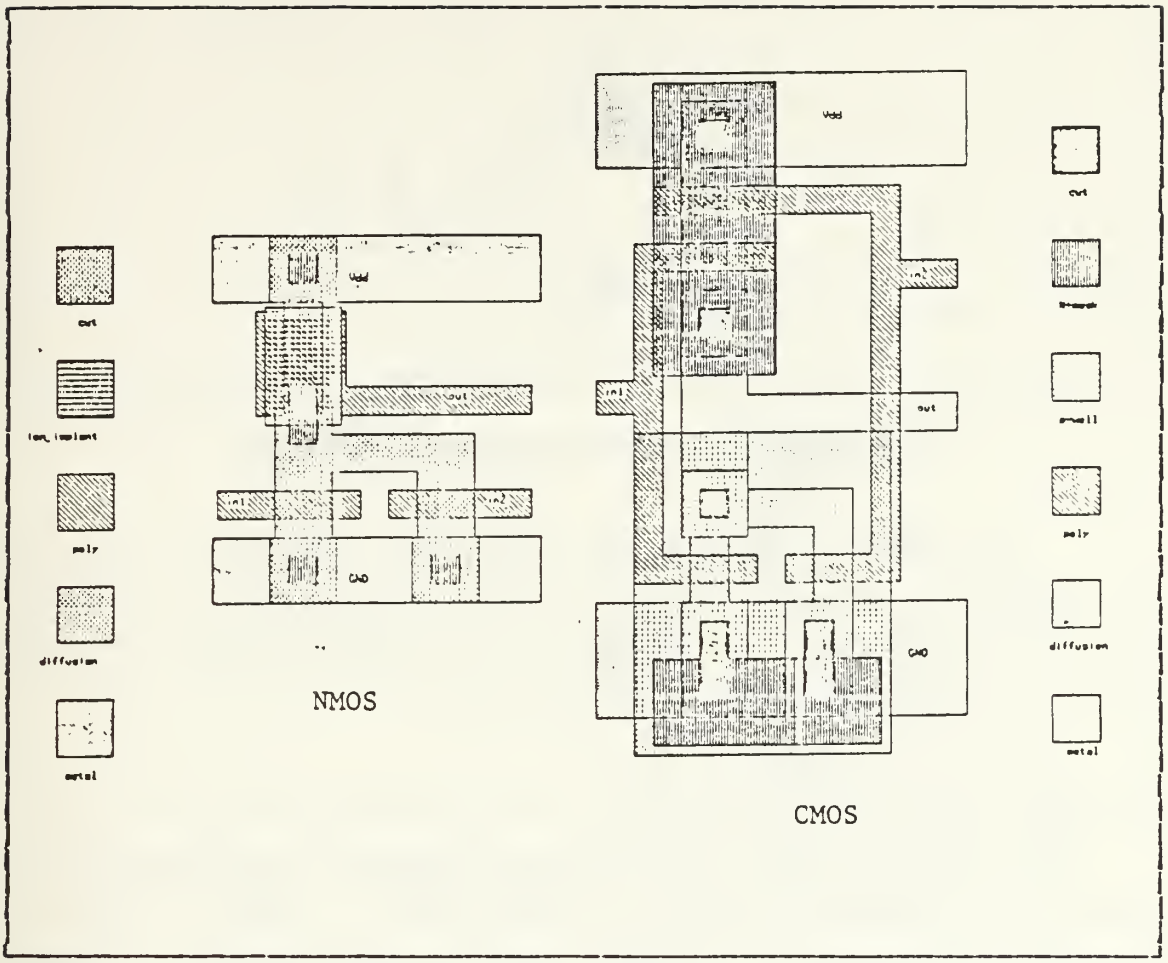


Figure 2.4 2-input Nor Gate.

high voltages well. The resulting unpredictable voltage drops make it necessary to utilize both types of transistors. This increase in complexity over its NMOS counterpart is partially offset by the absence of the level restoring circuitry NMOS requires following a pass transistor.[2]

---

[2]In NMOS digital circuits the length to width ratio of the pull down transistor is usually four times that of the depletion mode transistor load. This ratio is required to insure sufficient excursion of the output voltage. However, after a pass transistor is used, a ratio of 8:1 rather than 4:1 must be used to restore the VGS threshold voltage drop across the pass transistor.

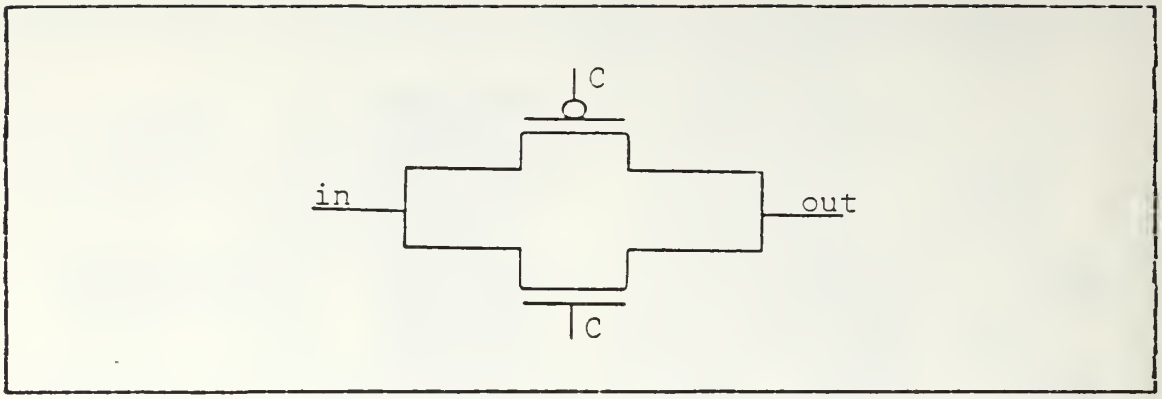


Figure 2.5 CMOS Transmission Gate.

In general CMOS technolcgies are ratioless. The use of "improper" ratios will not affect the logical operation of most CMOS gates, it will only affect the speed of operation of the gates.

B. CMOS DESIGN METHODOLOGIES

Static gate CMOS circuits have three serious deficiencies when compared to static NMOS gates. First, they are more area consuming. Second, they can be slower. Though the individual gates can be faster in CMOS, the p-channel and n-channel gates are in parallel, thus, the fanout[3] and the output load capacitance of each circuit are doubled Third, a CMOS static gate is redundant, duplicating its functionality in both the pull-up and pull-down section.

One approach to remedy these deficiencies is to use a static NMOS-like style of design as in Figure 2.6 Here the p-channel device is always on and the pull-up to pull-down dimension ratio is relied upon to produce the proper output voltage. This introduces power consumption problems and takes away the full excursion on the output. Another

---

[3]Fanout represents the number of transistors that the output of a logic gate must drive.

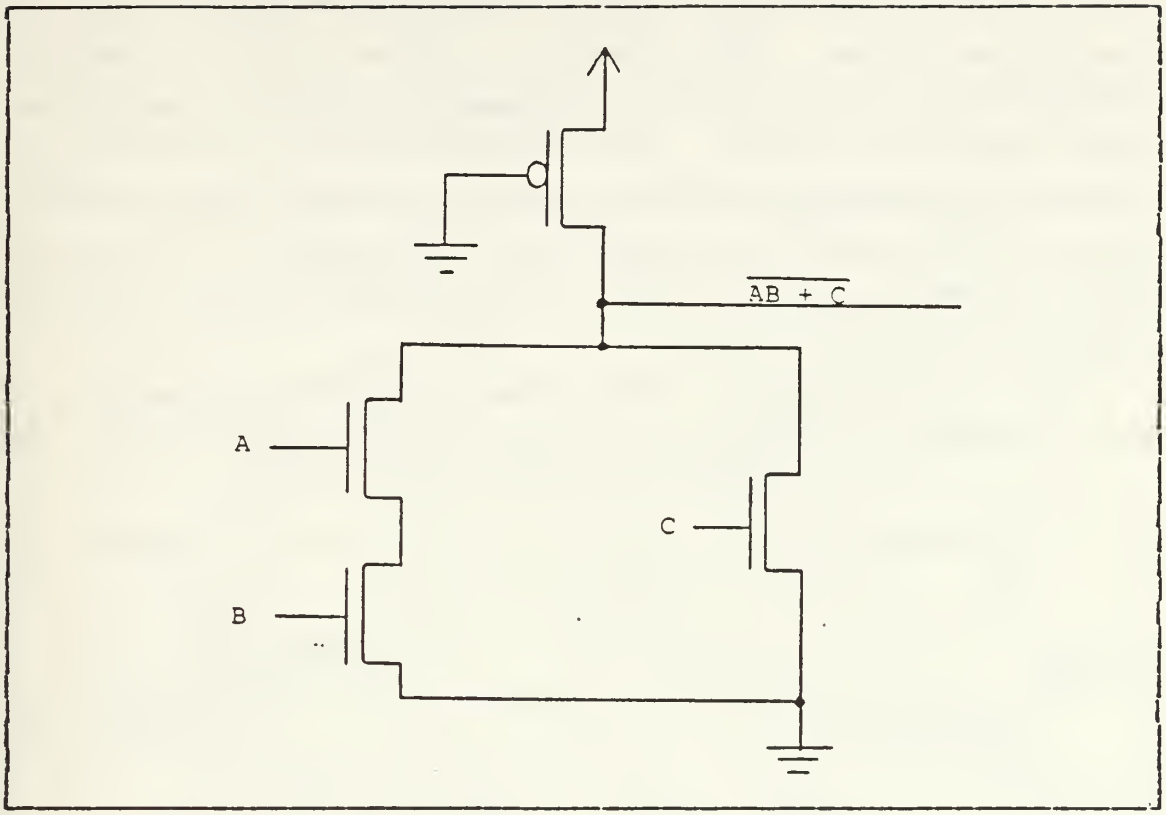


Figure 2.6 NMOS-Like CMOS Static Gate [Ref. 6].

approach is to make extensive use of transmission gates to build up logic functions. Using transmission gates means both polarities of all control signals are required. The resulting large number of wires required to route these control signals can become very area consuming, especially if only one metal layer is available.

A third and more effective solution is to use dynamic logic. Figure 2.7 contains three different implementations of a dynamic three- input NAND gate. In each, the output is meaningful (i.e. represents the value of the boolean expression in1 in2 in3) only when clk is high and $\overline{clk}$ is low. The circuits of Figure 2.7 (a) and (b) depend on the pull-up to pull-down ratio to produce the proper output. As with the NMOS-like style of design, full excursion on the output is

lost and there is steady state power consumption during the evaluation cycle. The circuit in Figure 2.7 (c) is precharged when clk is low and evaluation of the inputs takes place when clk is high. This configuration allows only one change of the output from 1 to 0, so the inputs must be stable at the time clk goes high. A change of one of the inputs from 1 to 0 after clk has gone high cannot cause the output to return to 1.

In general dynamic CMOS eliminates the redundancy of static CMOS by applying all inputs to one type of device and

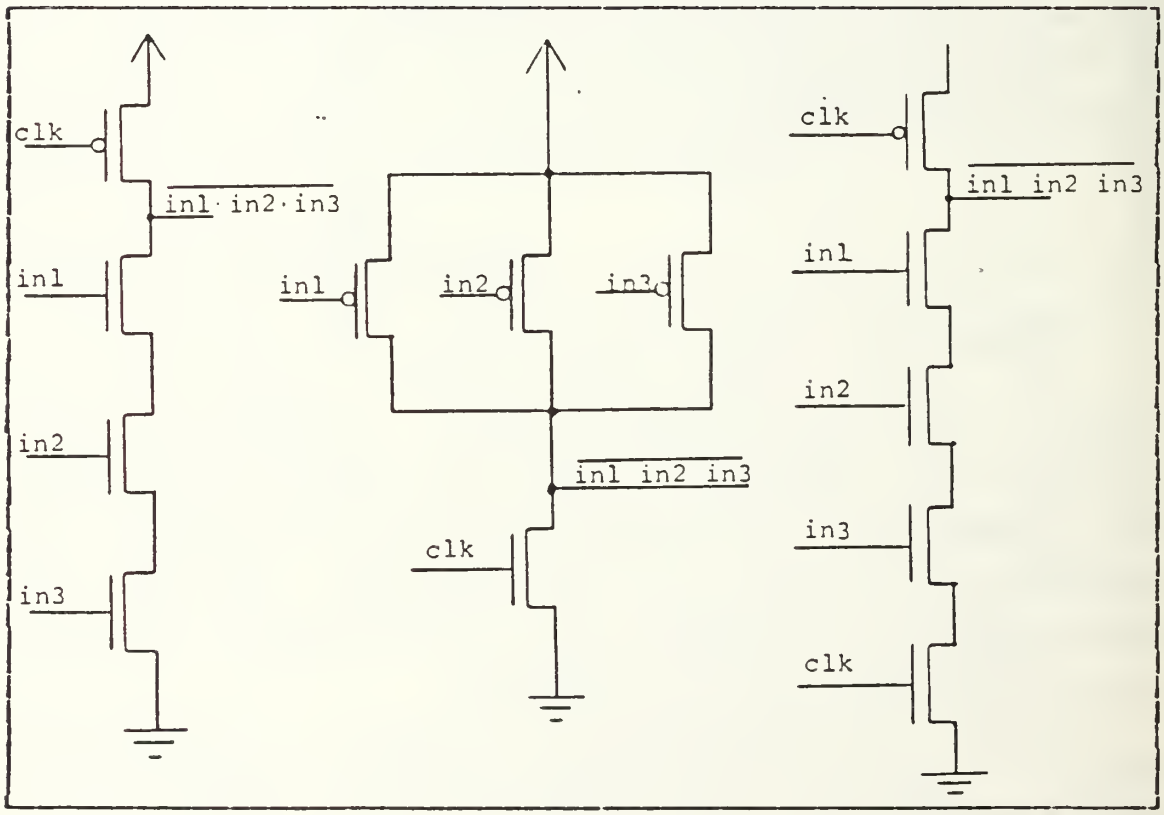


Figure 2.7 Dynamic NAND Gates [Ref. 6].

a control signal to the other type of device. The most popular dynamic CMOS logic design technique is domino CMOS [Ref. 7], illustrated in Figure 2.8 Here the output is the

logical AND of the boolean function (in1 in2 + in3) to be implemented and a control (clock) signal. When the clock is low, the circuit is precharged, and when the clock is high

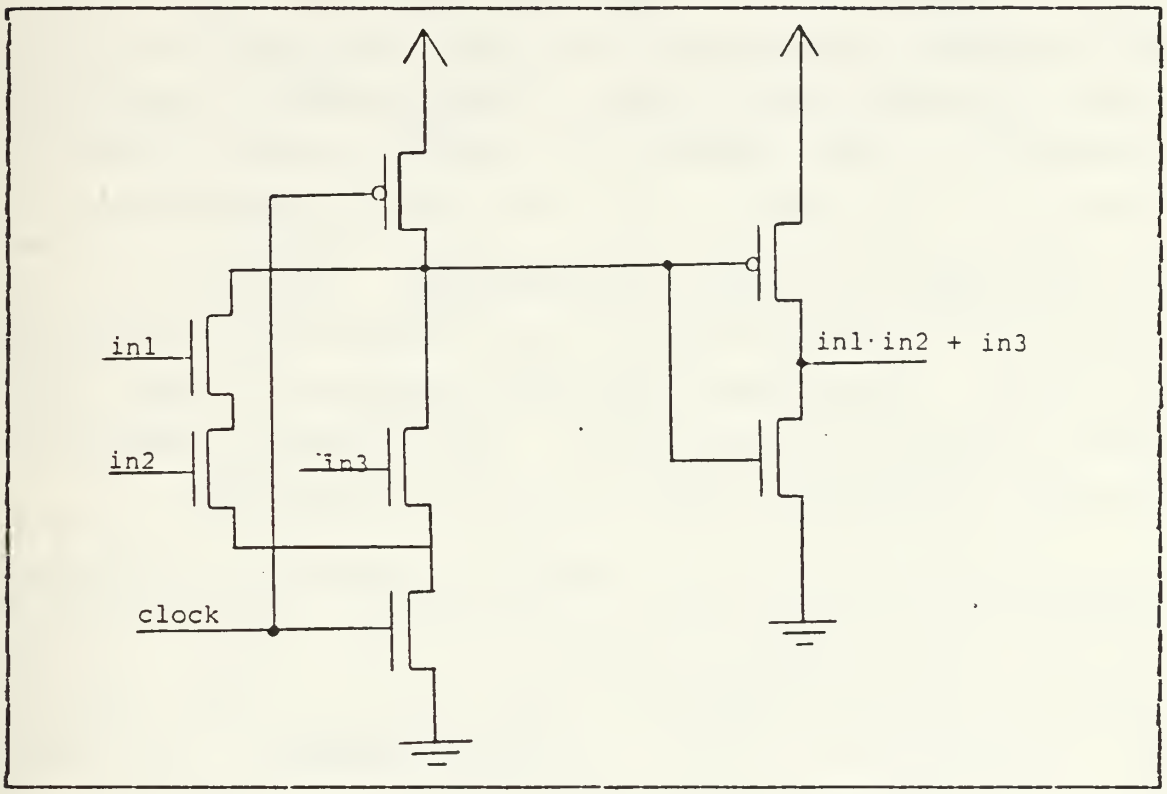


Figure 2.8 Domino CMOS Structure [Ref. 6].

evaluation occurs. With a common clock shared by all the domino gates on a chip, during the evaluation cycle the signals ripple through the chip as though the logic were purely static. The follow on inverter insures that the output of each gate is low when evaluation begins. This prevents the outputs of all gates from changing unless driven low by the inputs. Domino CMOS is not always the answer though. If the logic of Figure 2.9 were implemented in domino CMOS it would be more area consuming than the same circuit implemented in static CMOS. Dynamic CMOS is more

area consuming in this case because these are simple gates with only a few inputs. Each NCR gate if implemented statically would need two n-channel devices and two p-channel devices. If implemented dynamically, each NOR gate requires three transistors of one type (one for each input and one for the control signal) and one transistor of the other type (for the control signal again). The number of transistors needed remains the same but the dynamic logic requires the designer to keep three inputs electrically isolated instead of just two. And if the dynamic design technique is domino, six additional inverters will be needed. As can be seen in Figure 2.4, in CMOS a NOR gate can be constructed from just one stage. Adding the follow-on inverter of the domino design results in an OR gate. Thus a second inverter is required to return the logic to that of a NOR gate.

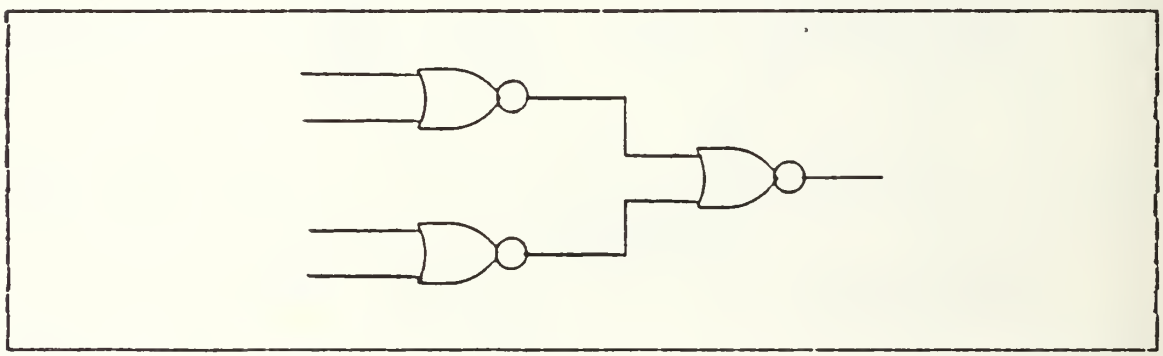


Figure 2.9 Circuit Difficult to Implement in Domino CMOS.

C. CMOS IMPLEMENTATION TECHNOLOGIES

One of the principal issues in the design of a process to implement CMOS digital circuits in silicon is how to isolate the two types of devices. This can be accomplished by using a completely insulating substrate or through a more complex fabrication process.

1. CMOS-SOS

The only process currently offered by Metal-Oxide Semiconductor Implementation Service (MOSIS) which uses an electrically insulating substrate is Silicon on Sapphire (SOS). In this technology the n-channel and p-channel transistors are formed on silicon islands left after etching an epitaxial layer of silicon on a sapphire (Al_2O_3) substrate.

2. CMOS-Bulk

The other CMOS processes offered by MOSIS all use CMOS-Bulk p-well technology. The p-well processes differ in the number of layers of metal interconnections (1 or 2) and the presence or absence of capacitors. In CMOS-Bulk p-well (n-well) the substrate is n-doped (p-doped) and the p-channel (n-channel) devices are in this substrate. To isolate the n-channel (p-channel) devices from the substrate a heavily doped p-well (n-well) is first placed to act as the back gate. The heavy doping of the p-well (n-well) degrades the performance of the n-channel (p-channel) device while the p-channel (n-channel) device is optimized. In p-well CMOS, though the mobility of electrons in the n-channel device still exceeds that of the holes in the p-channel device, the performance difference of the transistors is minimized. The more uniform performance of the two transistor types makes the p-well process appropriate for CMOS random logic.

Figures 2.10 and 2.11 represent the top and side views of the steps of the CMOS-pw process for the production of an inverter. These steps are: (1) starting with an n-type substrate the p-well is patterned, (2) The active areas in the p-well and on the substrate are established, (3) the polysilicon is patterned, (4) the two ion implant masks are placed (the N+ mask is simply the photographic

negative of the P+ mask) , (5) contact cuts are made, and (6) the metal is placed.

a. Latchup in CMOS-pw

One of the main problems associated with CMOS-Bulk, both p-well and n-well is latchup. Basically latchup involves generation of a short circuit between Vdd and GND, and can result in the complete destruction of a chip. Many researchers have tried to formally define the conditions [Ref. 8] that cause latchup to occur. This task is extremely complex because the phenomenon is so dependent on layout, which is unique to each chip design. Though a fully quantitative analysis of latchup is still not available, a qualitative analysis will show what happens on the chip when latchup occurs.

Looking at the side view of an inverter in Figure 2.12, parasitic bipolar transistors can be seen. The base of the npn transistor is the p-well and the base of the pnp transistor is the n-doped substrate. These parasitic transistors are connected as shcwn in Figure 2.13 . If the output of the gates goes below GND by a value equal to the threshold of the npn transistor, its emitter starts to inject current (electrons) intc the base (p-well) and the resultant collector current flows to the Vdd node. If the resistance between the Vdd ncde and the source of the pull-up p-channel MOS transistor, R1, is large enough, the voltage drop across R1 will exceed the threshold of the pnp transistor. The collector current (holes) of the pnp device flows to the GND node. If the resistance between the GND node and the source of the pull-down n-channel MOS transistor, R2, is great enough, the resultant voltage drop across R2 will increase the base current in the npn transistor. As is evident, there is positive feedback.

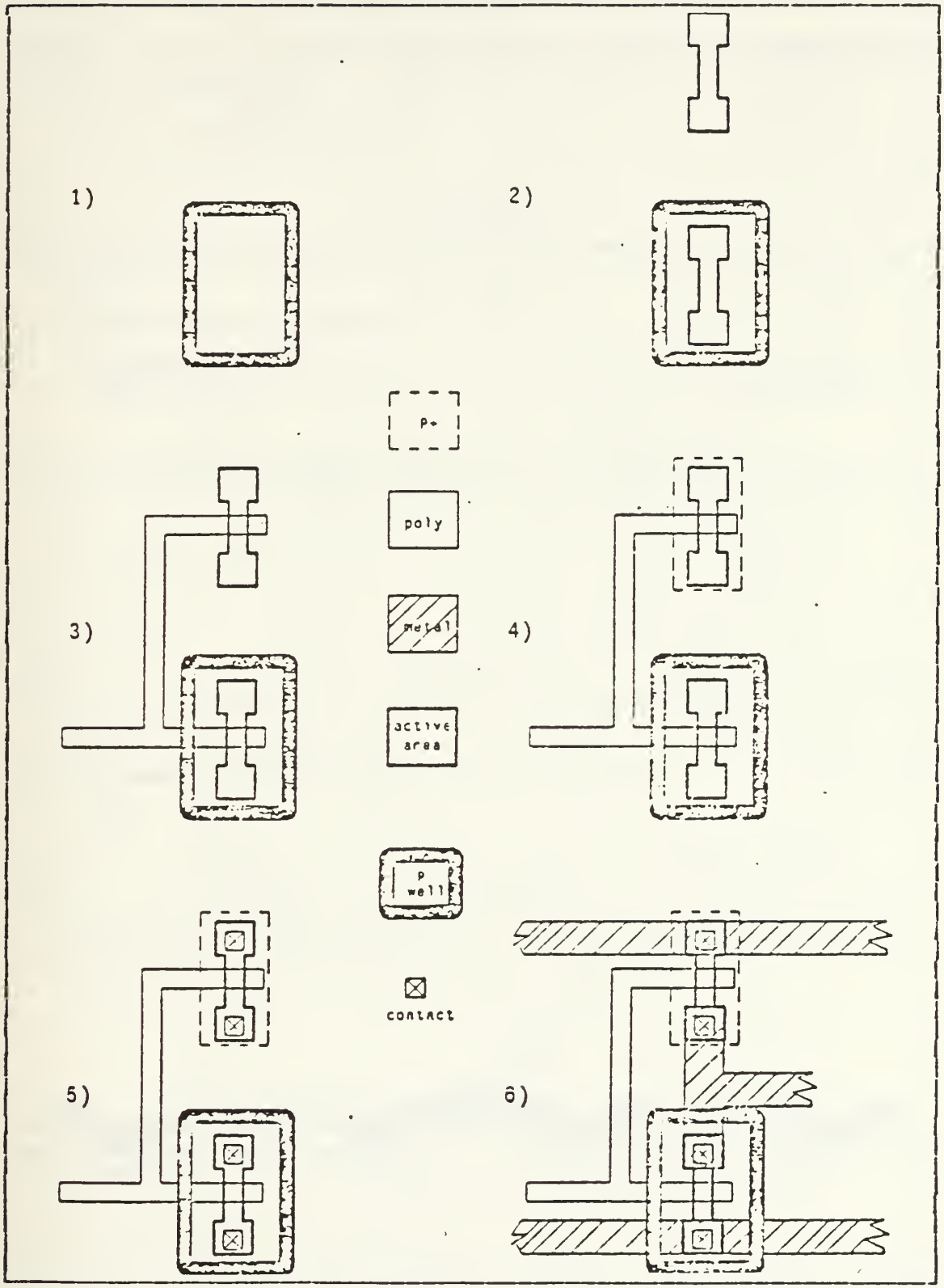


Figure 2.10 P-Well Process, Top View [Ref. 6].

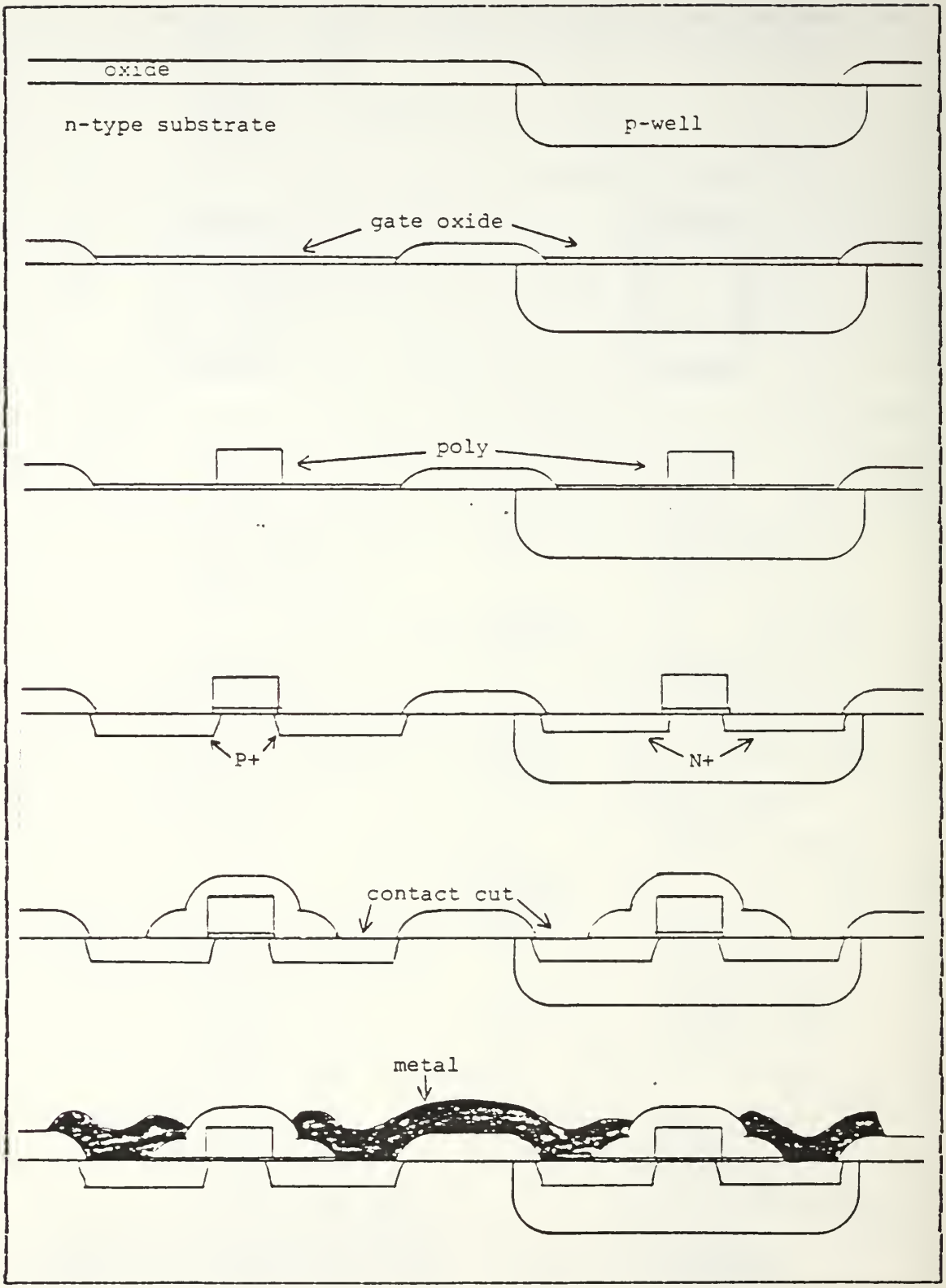


Figure 2.11 P-Well Process, Side View [Ref. 9].

The only way to stop this destructive process once it has started is to disconnect Vdd or GND. Prevention of latchup must be designed in.

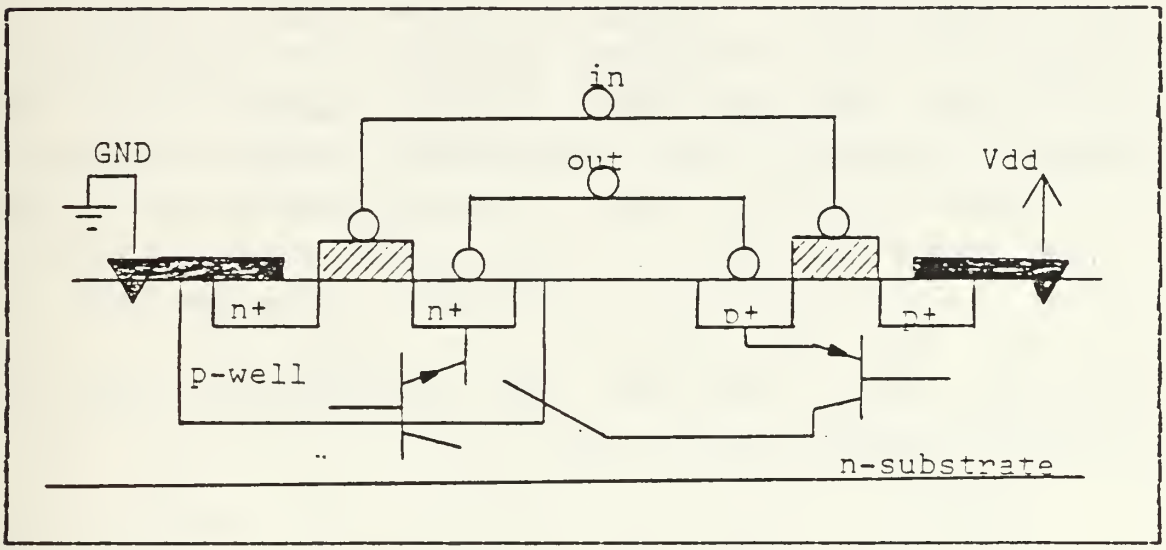


Figure 2.12 Bipolar Transistors in CMOS-Bulk [Ref. 6].

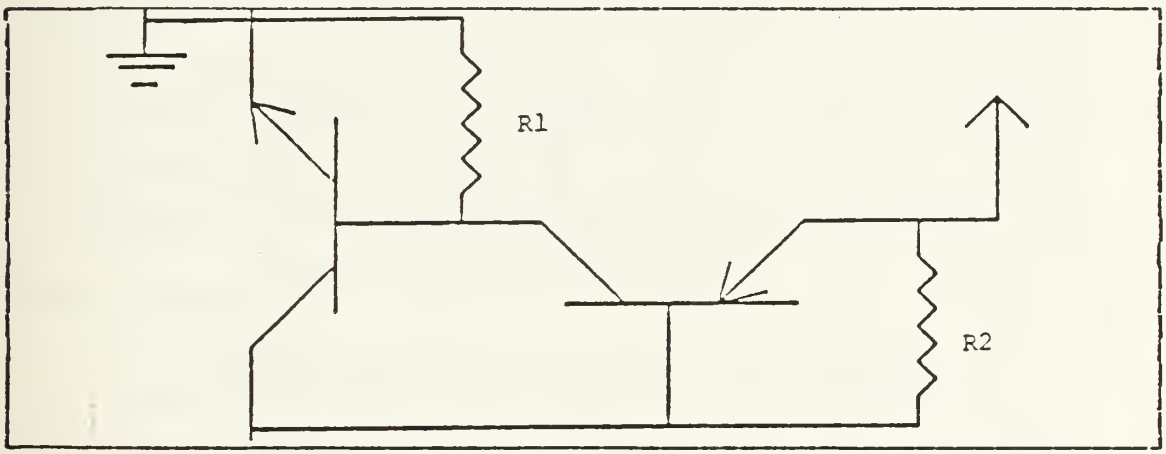


Figure 2.13 The Latchup Circuit [Ref. 6].

The MOSIS CMOS-Bulk p-well design rules include features for the specific purpose of reducing the

probability of latchup. The minimum separation rules for p-wells and P+ doped active areas exist for this purpose. Their aim is to reduce the gain of the parasitic bipolar transistors, thus requiring a larger noise spike of longer duration to start the latchup sequence. A frequently used technique is the grounding of the p-well as illustrated in Figure 2.14 . Here the effect of the P+ doped area covering half of the contact cut for the ground bus is to reduce the resistance R2 in Figure 2.13 . Another practice is to place a small capacitor across the Vdd and GND pins of CMOS-Bulk chips. To provide capacitive filtering of noise spikes on the chip, Vdd and GND busses are frequently run close together. Also, Vdd input pads are designed to provide capacitance between Vdd and GND.

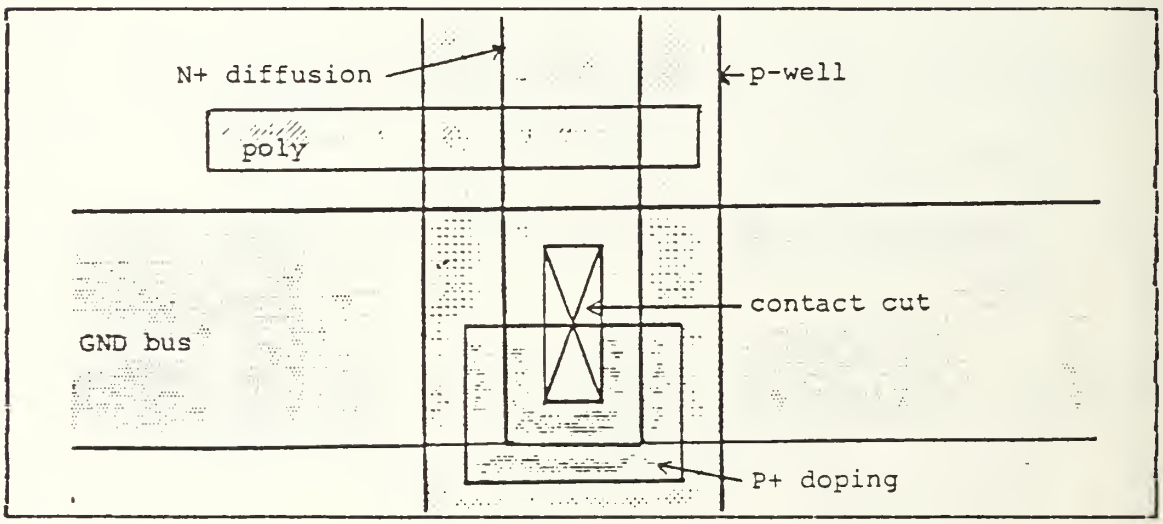


Figure 2.14 Grounding of the P-Well.

3. Twin-tub CMOS

This process, also called twin-well, uses both n-wells and p-wells on a high resistivity N- or P-

substrate, or in an epitaxial layer of silicon on a P+ or N+ wafer. Since the well doping does not have to overcome the substrate doping, both the n-channel transistors in the p-well and the p-channel transistors in the n-well can be optimized. Domino CMOS is enhanced by the use of this process since the optimized n-channel devices can speed up the complex boolean expression evaluation and the optimized p-channel devices can speed up the signal drive between stages (thereby reducing the effect of a given fanout).

D. CMOS TECHNOLOGY SELECTION

The CMOS implementation technologies available from MOSIS are CMOS-Bulk p-well with one metal layer, CMOS-Bulk p-well with two metal layers, CMOS-Bulk p-well with two metal layers and capacitors (for analog circuits) and CMOS-SOS.

The advantages of CMOS-Bulk are: (1) very good noise margin, (2) faster than NMOS, and (3) a proven reliable fabrication process. Its disadvantages are: (1) latchup susceptibility, (2) use of p-well guard rings is needed if radiation hardening is desired, (3) lower circuit density than NMOS or CMOS-SOS, and (4) more complex design rules than either NMOS or CMOS-SOS.

The advantages of CMOS-SOS are: (1) faster than NMOS or CMOS-Bulk, (2) very good noise margin, (3) intrinsically radiation hardened, and (4) no latchup. Its disadvantages are: (1) expensive fabrication process due to the sapphire, (2) sapphire variability reduces the reliability of the fabrication process, (3) thermal mismatch between the sapphire and silicon limits the carrier mobility, and (4) it is not a viable technology for dynamic memory due to back channel leakage.

CMOS-Bulk p-well was selected as the implementation process for the adder for the following reasons. First, technology files for this process were available at the Naval Postgraduate School (NPS) enabling the use of extant computer aided design (CAD) tools. Second, since this would be the first CMOS VLSI design at NPS, utilizing the most reliable process is prudent to prevent design problems from being clouded by implementation process problems.

III. DESIGN TOOLS

To employ the Mead-Conway design methodology on a large scale design, three computer aided design (CAD) tools are needed. A layout design editor for viewing the circuits as they are created is the first tool required. Next, a design rule checker is necessary to confirm that all the design rules for the specified technology have been adhered to. Though not a complex task, the large number of checks that must be made for even a modest design makes manual design rule checking highly error prone. Finally, a circuit simulator is needed to verify that the circuit as designed provides the proper logical output. In the design of the sixteen-bit pipelined adder, the Caesar layout editor [Ref. 4], the Lyra design rule checker [Ref. 10], and C. Terman's RNL circuit simulator [Ref. 11] were employed.

A. CAESAR

Caesar is a generic layout editor. It is not designed for any particular VLSI implementation technology. It is not even limited to designing integrated circuits. Caesar is a graphics layout editor for the creation and manipulation of rectangles where the user specifies the color, size, and placement. It is through the user specified technology file that the rectangles of color take on meaning. At the Naval Postgraduate School (NPS) there are two technology files available for use with Caesar. One is for N-doped metal oxide semiconductors (NMOS) and the other is for complementary metal oxide semiconductors utilizing a P-doped well (CMOS-pw).

Caesar works with files of its own special format. These file are indicated by an appended file type of ca (i.e. xxxx.ca). On command Caesar will generate a Caltech Intermediate Format (CIF) file of the same layout. Again it is the technology file which tells Caesar which CIF layer labels to attach to the colored rectangles.

At NPS, Caesar is set up to take commands from any terminal where the execution of the Caesar program is initiated (usually the ADM-3a console adjacent to the color graphics display unit) and from a four-button puck on a graphics tablet attached to the color display device. Caesar displays its graphics results on an AED 767 color monitor and displays its menus, messages, and prompts on the command console. Detailed information on the installation and operation of Caesar at NPS can be found in Reference 4 and Reference 2.

Caesar is an interactive CAD tool. The results of any command are rapidly displayed on the AED 767. The results of a command may be undone (u) or repeated (.) with a single stroke of the specified key on the command console. While running Caesar, a user may also call upon the design rule checker, Lyra, to check the area inside and within three Caesar units[4] of the current box for design rule violations. This interactive use of the layout graphics display and the design rule checker helps to insure that there will not be any design rule forced changes late in the design cycle when changes are much more time consuming. With Caesar's level of interaction with the designer, the design loop consisting of (1) issue commands to perturb existing circuit, (2) visual inspection to verify command's generation of desired

---

[4] A Caesar design is layed out on a grid of Caesar units. These units do not represent any specific length. When creating a CIF file from a Caesar file the desired length of a Caesar unit is specified.

results, and (3) design rule checking of new circuit, can be rapidly completed.

Caesar is a hierarchical design tool. With Caesar, circuits can be created by piecing together cells (other files of type .ca) which in turn may be made up of other sub-cells. Theoretically, there is no limit to the number of levels in the hierarchy. Not only can cells (sub-cells, etc.) be called upon to fill locations in a circuit, if they need to be modified to function properly, Caesar provides a subedit mode to facilitate editing of layouts one level below the current editing level. Care must be taken when this subedit feature is used since the changes made to the cell are global. Everywhere the given cell is used on the chip, the newly edited version will appear.

B. LYRA

Like Caesar, Lyra is a generic design rule checker. When Lyra is invoked from within Caesar, the actual program executed to check for design rule errors depends on the technology file indicated in the header of the Caesar file being edited. After running, Lyra sends a message to the command console indicating the number of errors found. On the graphics display Lyra paints the exact location of each error and labels each error with the design rule violated. The error label consists of abbreviations for the layers involved, followed by an underscore, followed by an abbreviation for the type of violation detected. Table 1 lists the abbreviations used by Lyra for CMOS-pw.

The winter 1983 distribution of the University of California at Berkeley (UCB) CAD tools included two versions of Lyra. One for the Mead-Conway NMOS design rules and the other for the Jet Propulsion Laboratory's (JPL) five-micron feature size CMOS-pw design rules. Since MOSIS no longer

TABLE 1

Lyra Error Abbreviations

Layer	Abbreviation		Error
polysilicon	p	w	minimum width
metal	m	s	minimum separation
p-well	w	x	malformed transistor
n+ diffusion	d		
cut	c		
p+ diffusion	P		

supports fabrication of the JPL CMOS-pw process, design rules for the MOSIS supported three-micron CMOS-pw process were obtained. Professor Marco Annatarone at Carnegie-Mellon University (CMU)generated the listing of the three-micron CMOS-pw design rules compatible with Lyra and has provided NPS with a copy. To generate executable code from the prototype Lyra program and imbed the specific process design rules, the program rulec (see Appendix B) is run with the design rule list file as its argument.

Now, when Lyra is invoked from Caesar while editing a CMOS-pw technology circuit, the three-micron minimum feature size CMOS-pw design rules are applied. This version of Lyra does not check for exceeding any maximum dimensions. The only maximum size design rule in this technology is for contact cuts, which may not exceed 3 microns by 8 microns. Avoidance of improper contact cuts can be accomplished by utilizing Caesar's hierarchical nature. Contact cuts of all needed sizes and types are generated once and saved to be inserted as cells wherever needed.

C. SIMULATION

Once a circuit layout has completed this initial design loop, it matches the designer's conception of how it should appear and is free of design rule violations. The performance of the given circuit, though, remains uncertain. To simulate the performance of the design, programs such as SPICE [Ref. 11] and RNL [Ref. 11] are used.

1. SPICE

SPICE is an important simulation tool in the design of high speed CMOS digital and analog circuits. With its detailed device modeling, SPICE can provide accurate predictions of performance once the device parameters of the implementation technology are known. SPICE provides the logical output of a circuit based upon the inputs and describes the transient behavior of the circuit as it changes to the new logical output. Thus SPICE enables a designer to optimize transistor dimensions for speed.

Unfortunately, the version of SPICE currently available on both the Vax 11-780 and the IBM 3033 at NPS (version 2G6) fails when the parameters of the devices fabricated by the MOSIS three-micron CMOS-pw process are used. With these parameters the transient behavior solutions do not converge.

Engineers at CMU, UCB, and the University of Washington (UW) are currently employing an experimental version of SPICE (version 2X.x developed at UCB) which is successful simulating with the three-micron CMOS-pw device parameters. This version, however, has other bugs and is therefore not available for general distribution. The changes to SPICE 2G6 that enable SPICE 2X.x to simulate the three-micron CMOS-pw devices will be incorporated into the next distribution of SPICE (version 2G7). The Naval Postgraduate School is in the queue of institutions to receive SPICE 2G7 once it is ready.

In order to run a SPICE simulation of a CMOS circuit designed using Caesar, the following steps should be executed. First, the labeling feature of Caesar is used to place labels on the electrical nodes of interest in the circuit (Vdd, GND, input, output, etc.). Second, the Caesar command

```
: cif 100 -p
```

is issued to generate the basename.cif file. The parameter 100 indicates a scale of 100 centimicrons per Caesar unit[5] and must be specified unless the default value of 200 centimicrons per Caesar unit is desired. The -p option causes entries to be made in the basename.cif file for the labels assigned. Third, after exiting Caesar and returning to Unix, the circuit extractor Mextra [Ref. 10] is invoked using the command

% mextra basename

to create the file basename.sim. To modify the basename.sim file to a SPICE file (basename.spice), the program sim2spice [Ref. 11] is used. The basename.spice file contains a list of transistors and capacitors in the circuit in a SPICE compatible format.

The basename.spice file must be edited to add the model parameters for the transistors, to specify the waveforms of the input(s), to specify the type of analysis to be performed (usually transient analysis) and to specify the output to be produced (tables, graphs, etc.). The Spice User's Manual [Ref. 11] contains the formats of these additions to basename.spice. Best case and worst case device model parameters for the MOSIS three-micron CMOS-pw process as compiled by Dr. M Annaratone of CMU and Dr. L. Glasser of MIT are found in Appendix A.

2. RNL

RNL is a timing and logic simulator for digital MOS circuits. It is an event driven simulator which uses a resistance-capacitance model of a circuit to estimate node transition times and to estimate the effects of charge

---

[5]Since the minimum dimensions for the 3-micron CMOS-pw process are specified in microns instead of lambda, CMOS-pw circuits are usually designed on Caesar using one micron per Caesar unit.

sharing.[6] After input values have been assigned by the user, RNL calculates the effects of those inputs by repeating the following operations until there are no further node value changes: (1) when a node is added to the network due to a transistor being turned on, the charge sharing implications of the new node's capacitance and logic state on each of its electrical neighbors is computed, (2) for each node that might be affected, Vthev and Rthev (the parameters of the Thevenin equivalent circuit) are calculated and the new logic state is determined from Vthev (0.0Vdd to 0.3Vdd = logic 0, 0.8Vdd to 1.0Vdd = logic 1, logic X otherwise), (3) if the node has changed state, the transition time is calculated using the node's capacitance, and (4) any changes are propagated to other nodes. Details of the computation methods used by RNL can be found in the RNL Version 4.2(UW) User's Guide [Ref. 11]. More important to the user is an understanding of what information RNL keeps, what it discards, and how it decides what to do next.

Basic to the operation of RNL is the idea of an event. The three elements of an RNL event are: (1) a node in the network, (2) a new logic state for the node, and (3) the time when the node value changes to the new logic state. RNL maintains a list of events, sorted by time, that tells what processing remains to be done. When the user changes an input, an event is added to the list. RNL sequentially processes the next event on the list, stopping when (1) the list is empty, (2) a node the user is tracing changes value, or (3) when the specified simulation time interval has elapsed. To process an event, RNL removes it from the list, changes the node's state to reflect its new value, and then

---

[6]Charge sharing refers to the capacitive effects that happen when two or more previously unconnected nodes, each having some charge and capacitance, become connected by a resistor (transistor turning on).

calculates any new events resulting from the node's new value.

In calculating new events, first all nodes that might be affected by the change are found and marked. This includes the source and drain of all transistors for which the current node is the gate and all nodes connected to these nodes through turned on transistors. The search through the network stops when a non-conducting transistor or an input is reached. For each marked node, two calculations are made. First, a charge sharing calculation is performed to model changes of state due to the charging and discharging of node capacitances. Second, a final value calculation is done to determine the node's ultimate logical state.

A given node can have only two events pending: (1) a charge sharing event describing an immediate change in the node's state due to charge redistribution among the nodes on the connection list, and (2) a final value event describing the final, driven state of the node. RNL observes the following rules for processing events: (1) when a new charge sharing event is scheduled, throw away all previously pending events for the node, and (2) when a new final value event is calculated, it will be ignored if (a) there is a pending final event for the same value which is scheduled to occur sooner, (b) there is a pending charge sharing event for the same value as the new final event, or (c) there is no charge sharing event and the new final value event is the same as the node's current value. These rules are based on the assumption that the event that was last calculated reflects the latest configuration of the network and therefore should override events calculated earlier. Charge sharing events discard any pending final value events because any charge sharing calculation is immediately followed by a new final value calculation.

These event rules, however, sometimes lead RNL to generate incorrect results. This is especially true of signal driven circuits (circuits where inputs are applied to the source and drain of a transistor as well as its gate) and circuits that depend on the analog properties of the devices to predict the behavior of the circuit. For example, consider the first exclusive OR gate design for the

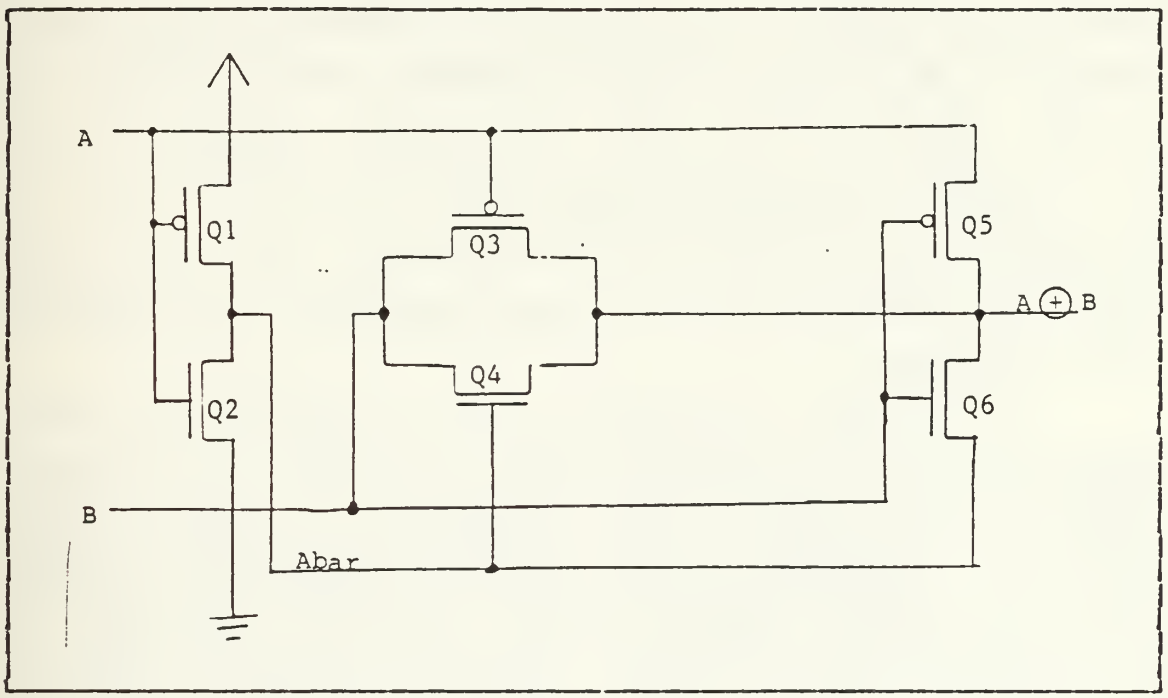


Figure 3.1 CMOS Exclusive OR [Ref. 6].

pipelined adder in Figure 3.1 This design has proven to function correctly at CMU, however, the RNL simulation shows this circuit failing.

Starting in a state where A=0, B=1, and out=1, assume that the input A then transitions to 1. Initially Q1, Q3, Q4, and Q6 are on. When input A goes high, Q3 is turned off (no events generated) and Q2 is turned on, generating a charge sharing event and a final value event for

Abar resulting in Abar going low. When Abar goes low, the still turned on Q6 is now trying to drive the output node low and the still turned on Q4 (RNL recognizes that it takes a finite amount of time for Q4 to turn off but does not recognize that n-channel transistors do not conduct high voltages well) is still trying to drive the output node high. The result is an output of X, the undefined state. Next, Q4 is turned off. Since turning off Q4 adds no new nodes to the network, the event list is empty and the output remains at X. The primary difficulty RNL has with this circuit centers around the fact that the output node is controlled by two nodes that can change at different times. As a result, a charge sharing event due to one input can eliminate a final value event of the other, with that final value event being the force which determines the circuit's actual behavior.

The circuit of Figure 3.2 is a proven latch design which also fails in RNL simulation. In Figure 3.2 the fractions next to the transistors represent the length to width ratios of the devices. This circuit is dependent on these ratios for proper operation. These ratios insure that the gain of the input signal on the gates of Q5 and Q6 is greater than the gain of the feedback signal to the same gates. RNL does not recognize the difference in these gains to be sufficient to cause the gates of Q5 and Q6 to be at either logical 1 or 0 when the input signal is the opposite of the feedback signal. As a result, the circuit becomes locked up at X. Because of RNL's difficulty with these two circuits, other designs were employed in the final adder (see chapter 5) to facilitate testing of the overall design.

To use RNL as installed at NPS, the following steps should be followed. First label the circuit and generate basename.cif as before. Again the program Mextra is used to extract the circuit, this time with the -o option (Mextra

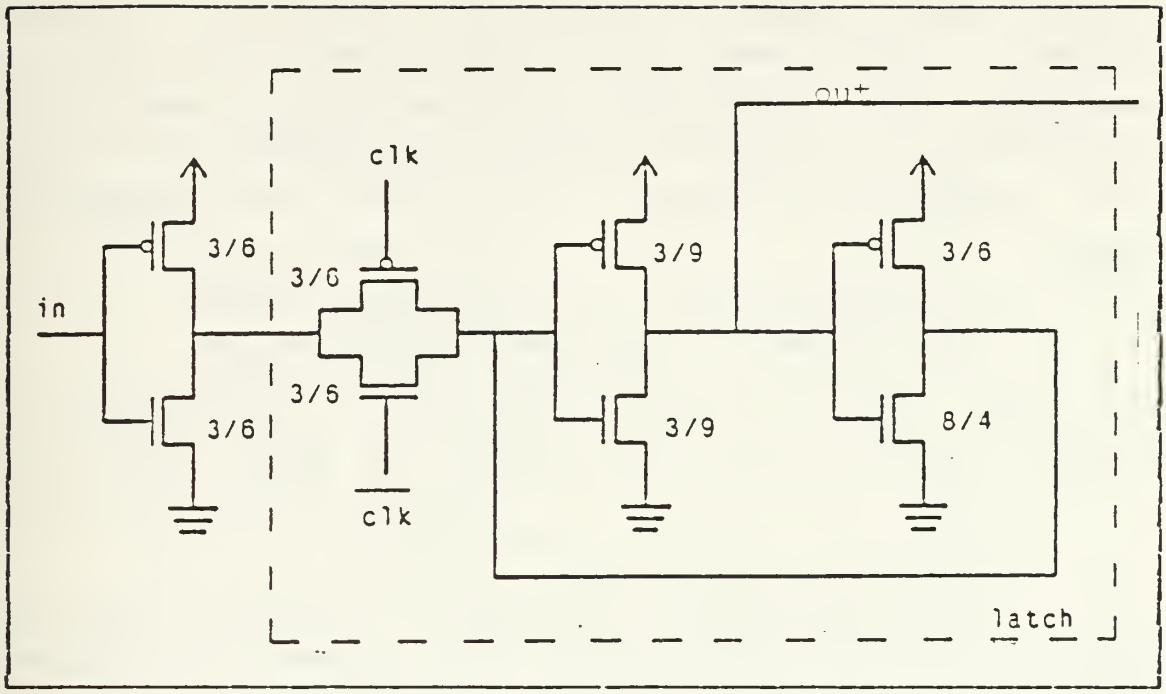


Figure 3.2 CMOS Latch Design [Ref. 6].

basename -o). The -o option causes Mextra not to compute capacitances. A follow on program in this sequence, Presim, performs this computation with greater accuracy. It should be noted that there are three different circuit extraction programs, each named Mextra. There is the MIT version, the UCB version and the UW modified UCB version. The next tool to be used in the sequence, Presim, can accept the output format of the MIT version and the UW modified UCB version. At NPS, the UCB version is installed and was used. The MIT and UW modified UCB versions differ in the order of the parameters in a transistor specification. Professor Annaratone at CMU developed a program, cformat, to change a .sim file generated by the UCB version to the MIT format. However, cformat does not work if the -o option is used with Mextra. To avoid a loss of accuracy, the .sim file can manually be changed to the UW modified UCB format. The

first step in this format change is to use the VI text editor to add "format: UCB" to the header line of basename.sim. The other change that needs to be made is to change the labels for the n-channel transistors from "n" to "e". Using the EX editor, the following steps accomplish this:

```
% e basename.sim      - invokes the editor
: g/ n/s//e/g         - make global change
                        for all n as first char
                        in a line, change to e
: w                   - write back edited file
: q                   - exit editor
```

The next step is to create a binary file for RNL from basename.sim using Presim. This is done by issuing the command:

```
% presim basename.sim basename config
```

Basename.sim is the edited .sim file and basename is the file into which presim writes its binary output. Config is the calibration file used to select other than default values for the circuit element capacitance and resistance. A copy of the presim user's guide from the UW/NWC VLSI Consortium release 2.0 and the calibration file used in simulating the adder are contained in Appendix C. The values used in the calibration file are taken from the MOSIS supplied electrical parameters.

The final step is to run RNL itself. This is done by entering one of the following two Unix commands:

```
% rnl        or
% rnl cmdfile
```

where cmdfile is the name of a file containing a sequence of RNL commands. Entering the first Unix command will cause RNL to take its commands directly from the console

interactively. If the second Unix command is used, specifying a command file, RNL first executes all the commands in cmdfile and upon completion, starts taking commands from the console. In either case, RNL should be given the following commands:

```
(load "uwstd.l")
(load "uwsim.l")
(read-network "basename")
```

where basename is the file generated by presim. The first two commands load RNL with several macros which simplify user interfacing with RNL.

The user interface with RNL is a LISP interpreter. The interpreter continuously executes the loop: (1) read a command, (2) evaluate the command and perform the specified actions, and (3) print the result. There are two formats for specifying commands to this loop. The first is:

```
(function argument argument ... argument)
```

Here the parentheses delimit the command and spaces separate the elements. The interpreter reads the entire command, up to the closing parenthesis, then the first element is interpreted as a function and all the others as arguments. The arguments may be of the same command form, (function arg arg ... arg). If the following command were issued to RNL,

```
(*  12  ( + 2 2)    (/  14  7 ))
```

RNL would respond by typing 96 (12*4*2). The other format for commands to RNL is

```
(function '(argument argument ... argument))
```

where the " ' " indicates the quote special form which keeps its argument from being evaluated. For example, (+ 2 3) evaluates to 5, but '(+ 2 3) is a string of three elements. When this second RNL command format is not used to represent an argument of another command (i.e. is not contained within

the parentheses of another command), it may be written in the more natural form:

function argument argument .... <newline>

Tutorials on RNL are contained in the University of Washington/Northwest VLSI Consortium's VLSI Design Tools Reference Manual [Ref. 11]. There are two points concerning the Mextra, Presim, RNL simulation cycle a user should be aware of that are not brought out in the documentation. The first concerns the use of vectors in RNL commands. As evidenced in the tutorials of Reference 11 and the adder simulation results in Appendix D, vectors can be used to make the input and output of RNL less cumbersome and verbose. After the vector has been defined, a user will then want to assign values to it. The documentation shows the format of the vector value assignment command to be:

(invec '(vecname values))

However, the "values" field has its own specific format. The first character should be a 0 or a 1 indicating positive and negative numbers, respectively. The LISP interpreter will work with negative numbers but RNL will not accept negative numbers as logical inputs. The second character is a letter specifying the number base of the input vector (b for binary, h for hexadecimal). For example, to assign the binary value +101010 to the vector vectone, the RNL command would be:

(invec '(vectone 0b101010))

The other point concerns the location of input labels on the input pads. When the entire chip is being simulated, the input labels are normally placed on the metal pads where the off chip leads are attached. Before an input signal from a bonding pad reaches the interior circuits of a chip it must pass through a resistor in an overvoltage

protection circuit. In the extraction and simulation process this resistor is viewed as an open circuit. Therefore, on input pads, the input label must be placed after the resistor in the signal path.

With Caesar, Lyra, and RNL, a designer at NPS has the requisite CAD tools for the complete logical circuit design loop. With these tools circuits that are free of design rule errors and produce the desired logical results can be designed. The lack of SPICE somewhat restricts the designer's ability to optimize speed, but there are several design techniques that can be employed to design chips that run fast. These will be covered in the next chapter.

IV. DESIGN OF THE ADDER

As stated in the introduction, the primary goals of the adder design are to maximize throughput and to provide for testability. The adder is to be a pipelined adder. Every clock cycle it should accept as inputs two 16-bit addends (A1, the least significant bit, through A16 and B1, the least significant bit, through B16) and one carry-in (Cin) bit. It is desired to produce the 16-bit sum (S1 ,the least significant bit, through S16) and the carry-out (Cout) bit as quickly as possible. Both the number of clock cycles from input of the addends to the output of the sum and the duration of each clock cycle are to be minimized. A secondary consideration in the design is expandability. An expandable design is one that can easily be extended to produce a 32-bit or 64-bit sum utilizing the same circuit structures. In this chapter the logical design and layout design of the 16-bit adder will be presented. The equations presented in this chapter are taken or derived from equations found in chapters three through six of The Logic of Computer Arithmetic by Flores [Ref. 12]. In these equations concatenation implies the logical AND, the symbol + implies the logical OR, and the symbol + implies the logical XOR.

A. LOGICAL DESIGN

In considering the speed spectrum of adders from a logical standpoint, at the fast end there is the table look-up. With 33 binary inputs and 17 outputs, this would require an address space of 2^{33} 17-bit words. With current technology this is not feasible. At the other end of the spectrum is the serial adder. On clock cycle 1 it uses A1,

B1, and Cin to produce S1 and C1out (carry out of bit one into bit 2). On clock cycle 2 it uses A2, B2, and C1out to generate S2 and C2out. Here 16 clock cycles elapse before the sum is available. An adder can also be implemented as a ripple carry adder where the duration of each clock pulse is sufficient to allow a carry into the sum to propagate all the way through to a carry out. In the case of the 16-bit adder, this would require a clock duration at least sixteen times the length of the gate delay of the one bit adder. The middle ground belongs to the carry look-ahead adder [Ref. 3]. In carry look-ahead (CLA) addition the carry into each bit position, C(i), is generated from the propagate,

$$P_{(i)} = A_{(i)} \oplus B_{(i)} \qquad \text{(eqn 4.1)}$$

$$G_{(i)} = A_{(i)} B_{(i)} \qquad \text{(eqn 4.2)}$$

P(i), and generate, G(i), primitives. P(i) = 1 implies that a carry into bit(i) will be propagated through to bit (i+1). G(i) = 1 implies that A(i) and B(i) will provide a carry into bit(i+1) of the sum, regardless of the contents of the

$$C_{(i)} = G_{(i-1)} + G_{(i-2)} P_{(i-1)} + \cdots + C_{in} P_{(i-1)} \cdots P_{(2)} P_{(1)} \qquad \text{(eqn 4.3)}$$

$$S_{(i)} = C_{(i)} \oplus P_{(i)} \qquad \text{(eqn 4.4)}$$

less significant bits of A and B. The algorithm for the CLA sum generation is as follows. The first event is the evaluation of equations 4.1 and 4.2 to generate the P(i) and G(i) primitives. The second event uses the P(i) and G(i) primitives as inputs to equation 4.3 to generate the C(i)'s. The final event is the computation of the S(i)'s from equation 4.4 .

As pointed out by Flores [Ref. 12] and by Conradi and Hauenstein [Ref. 3], there are several logical implementations of carry look ahead addition. A principal task of this thesis investigation was to select a fast logical design. Without the circuit simulator Spice, the analysis of each design considered was more qualitative than quantitative. In this qualitative analysis, a turned on transistor is considered as a resistor with its resistance proportional to its length and inversely proportional to its width. All gates driven by such a turned on transistor are considered to be capacitive loads with capacitance proportional to the area of the gate. The interconnect wiring is considered to add both parallel capacitive loading and series resistance as shown in Figure 4.1

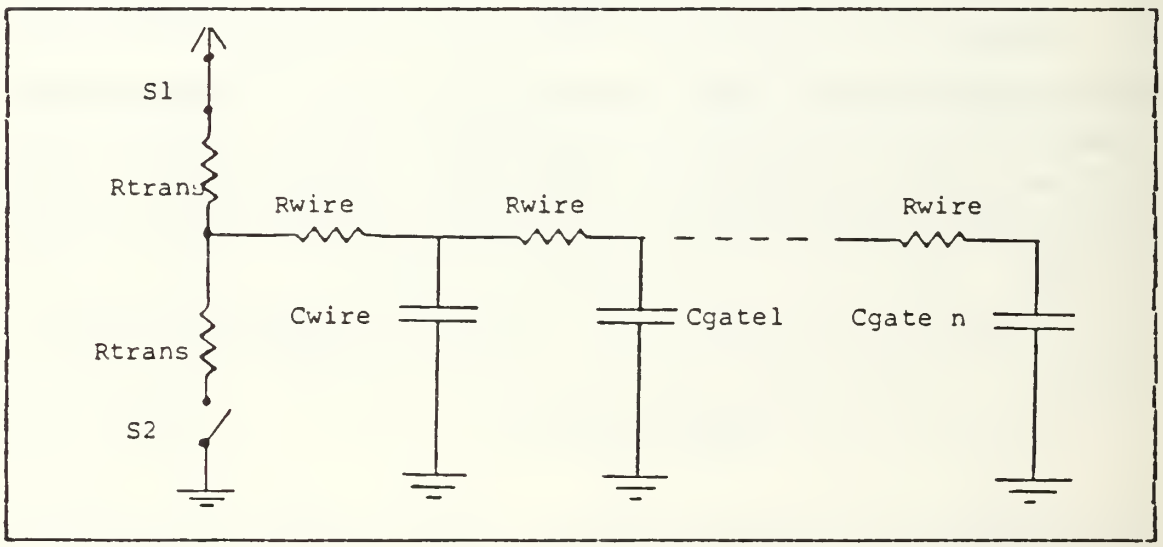


Figure 4.1 CMOS Output Loading Model.

From this model it is obvious that the amount of interconnect wiring and the number of gates driven (fanout) should be minimized to minimize the output transition time when the positions of switches S1 and S2 of Figure 4.1 are

reversed. This led to the following guidelines in the design of the adder:

1) The internal logic of each stage should be acccm-plished with minimum dimension transistors , 3 microns x 4 microns (length x width). This leads to more compact circuits with shorter interconnections and reduces the capacitive load on the preceding stage.
2) Significantly wider transistors (3-micron x 9-micron) should be used at the output of each stage where the fanout and interconnect loading is greater.
3) The fanout of any transistor should be kept to less than five.

This requires a more complete definition of fanout because the capacitive loading of a gate depends on its area. A 3-micron x 4-micron transistor driving six other 3-micron x 4-micron transistors has a fanout of six. A 3-micron x 8-micron transistor driving the same load is considered to have a fanout of three. Though this implies that a high fanout problem can be solved by merely increasing the width of the driving transistor, it neglects the effects of the interconnect wiring. As gates are added to the load of a transistor, each subsequent addition must be more remote from the driving transistor. Since the resistance of the wiring is prcportional to its length and inversely proportional to its width, the resistance of the wiring will increase unless the width is also increased. However, since the capacitance cf the wiring is proportional to its area, most of the gain achieved by widening the wire to reduce resistance is offset by the increase in capacitance. As a result, in the design of the adder, increasing the width of the driving transistor was not viewed as a complete fix for a fanout problem.

For the comparison of the different approaches to CLA addition, the term logical event needs to be defined. The

most basic definition is a combinational logic circuit accepting a set of inputs, performing its specified operations on those inputs and generating a set of outputs. Therefore, the input of the addends, followed by the computation and output of the sum can be considered as a logical event. However, a primary design consideration for the adder is to provide for testability and a key element of this provision is the availability of intermediate results (see section B of this chapter). This implies breaking up the sum generation into several separate events. The first event takes the addends as inputs, performs some logic operation(s) on them and stores the results in a register. The next event takes its inputs from that register and stores its results in another register. This chain continues until the last event deposits the sum on the output pads of the chip. To provide the tester with easily interpreted intermediate results, the equations presented in this chapter were taken as boundaries for each logical event. The terms on the right side of the equation determine the inputs and the left side terms determine the output of a logical event. Once all the inputs for an equation are generated by previous events, the logic of the equation becomes part of the current event.

1. Zero Level CLA Logic

This logic requires three events to generate the sum. First, equations 4.1 and 4.2 are used to generate the P(i)'s and G(i)'s. Second, from equation 4.3 the C(i)'s are generated. Finally, the sum is derived from equation 4.4 The principal problem with this approach for a sixteen-bit adder lies in the application of equation 4.3 Here, the input P(1) has a fanout of 15, which makes this approach unsatisfactory.

2. First Level CLA Logic

Noting that a four-bit sum generated using zero level CLA logic is within the design guidelines suggests cascading 4-bit slices of the same logic as indicated in Table 2 Here the sum is available after six events and the

TABLE 2
First Level CLA Logic for a 16-bit Sum

Event No.	Bits 1-4	Bits 5-8	Bits 9-12	Bits 13-16
1	Compute P(i),G(i)	Compute P(i),G(i)	Compute P(i),G(i)	Compute P(i),G(i)
2	Compute C(i)	Delay P(i),G(i)	Delay P(i),G(i)	Delay P(i),G(i)
3	Compute S(i)	Compute C(i)	Delay P(i),G(i)	Delay P(i),G(i)
4	Delay S(i)	Compute S(i)	Compute C(i)	Delay P(i),G(i)
5	Delay S(i)	Delay S(i)	Compute S(i)	Compute C(i)
6	Delay S(i)	Delay S(i)	Delay S(i)	Compute S(i)

fanout is reduced by a factor of four. The event cycle time reduction would more than make up for the event count increase since cycle time grows faster than linearly with fanout. The only drawback with this design lies in the cost of extending it to generate 32-bit or 64-bit sums. For every 4-bit slice added, another event is required. Thus, a 64-bit add would require 12 events.

3. Second Level CLA Logic

Again the data is divided into 4-bit slices called blocks. But rather than let the carries ripple through the blocks, two new primitive functions are introduced. They

are the block propagate, BP(i), and block generate, BG(i), functions. BP(i)=1 implies that a carry into block(i) will be propagated through to block(i+1). BG(i)=1 implies that block(i) will generate a carry into block(i+1). For a 4-bit block where bit(1) is the least significant bit, The BP and BG primitives are generated by equations 4.5 and 4.6 respectively, with the P(i)'s and G(i)'s computed as before.

$$BP_{(i)} = P_{(4)}P_{(3)}P_{(2)}P_{(1)} \qquad \text{(eqn 4.5)}$$

$$BG_{(i)} = G_{(4)}+G_{(3)}P_{(4)}+G_{(2)}P_{(4)}P_{(3)}+G_{(1)}P_{(4)}P_{(3)}P_{(2)} \qquad \text{(eqn 4.6)}$$

Next, the block carry, BC(i), which represents the carry from block(i) into block(i+1), is computed using equation 4.7 which represents the same logic as equation 4.3

$$BC_{(i)} = \sum_{k=0}^{i}\left[BG_{(k)}\prod_{j=k+1}^{i-1}BP_{(j)}\right] \qquad \text{(eqn 4.7)}$$

So far, after three events, the P(i)'s, G(i)'s, BP(i)'s, BG(i)'s, and BC(i)'s have been generated. If the same method of generating the final sum as used in zero level CLA were to be used, two additional events would be required. The first again applies the logic of equation 4.3 to each 4-bit block to generate the carry into each bit. Here the Cin for block(i) is given by BC(i-1). The second cycle is used to generate the sum from the C(i)'s and P(i)'s. One of these events can be eliminated if, while the BC(i)'s and their predecessors are being computed, an estimated sum of the 4-bit block is also computed. One method is to compute two estimated sums for each block, one assuming an carry into the block of 0 and the other assuming a carry in of 1. When the correct carry in for block(i) is generated, it is used to multiplex the correct sum for the block to the output. This assumed carry method was rejected

because of the large amount of area consumed by the registers needed to hold two possible answers. The second method is to compute the estimated sum of the block assuming a carry-in of 0 and then correcting the estimated sum once the actual carry-in to each block is known.

Since the estimated sum, ES(i), is not needed until after the third event and computing it as one event again leads to fanout problems, the computation of ES(4), the most significant bit, through ES(1) is computed in two events as follows. First, an intermediate estimated sum, IES(i), is computed using two-bit slices, each assuming a 0 carry in (see equations 4.8 through 4.11). At the same time, a carry from bit(2) into bit(3) (IC23) is computed using equation 4.12 On the next event, ES(i) is computed from the IES(i)'s and IC23 using equations 4.13 through 4.16 .

$$IES_{(1)} = P_{(1)} \qquad \text{(eqn 4.8)}$$

$$IES_{(2)} = P_{(2)} \oplus G_{(1)} \qquad \text{(eqn 4.9)}$$

$$IES_{(3)} = P_{(3)} \qquad \text{(eqn 4.10)}$$

$$IES_{(4)} = P_{(4)} \oplus G_{(3)} \qquad \text{(eqn 4.11)}$$

$$IC23 = G_{(2)} + G_{(1)} P_{(2)} \qquad \text{(eqn 4.12)}$$

$$ES_{(1)} = IES_{(1)} \qquad \text{(eqn 4.13)}$$

$$ES_{(2)} = IES_{(2)} \qquad \text{(eqn 4.14)}$$

$$ES_{(3)} = IC23 \oplus IES_{(3)} \qquad \text{(eqn 4.15)}$$

$$ES_{(4)} = \left[IES_{(3)} IC23 \right] \oplus IES_{(4)} \qquad \text{(eqn 4.16)}$$

Now, after three events, estimated sums for each 4-bit block and the actual carry into each block (Cinb) are available. From these the sum can be computed using equations 4.17 through 4.20 .

$$S_{(1)} = C_{inb} \oplus ES_{(1)} \qquad \text{(eqn 4.17)}$$

$$S_{(2)} = \left[C_{inb}\, ES_{(1)}\right] \oplus ES_{(2)} \qquad \text{(eqn 4.18)}$$

$$S_{(3)} = \left[C_{inb}\, ES_{(1)} ES_{(2)}\right] \oplus ES_{(3)} \qquad \text{(eqn 4.19)}$$

$$S_{(4)} = \left[C_{inb}\, ES_{(1)} ES_{(2)} ES_{(3)}\right] \oplus ES_{(4)} \qquad \text{(eqn 4.20)}$$

Using second level CLA logic, the 16-bit sum is generated in only four events. Additionally, this design can easily be extended to the generation of 64-bit sums. The logic of equations 4.5 and 4.6 which produced the second level primitives BP and BG can be used again to generate third level primitives, B3P and B3G. These third level primitives represent the carry propagate and carry generate properties of 16-bit slices. The carry into each 16-bit block is provided by implementing equation 4.7 . Thus, adding one event will provide the carry into each of four 16-bit blocks of a 64-bit sum. The logic of equation 4.3 is then used to generate the carry into each 4-bit block of the sum and the final sum is computed as before. The final result is that by adding two events, for a total of six, and using the same logic as before (i.e. no new circuits need to be designed), the 16-bit adder can be extended to a 64-bit adder.

B. DESIGN FOR TESTABILITY

Another primary objective of the adder design was to provide for testability, that is, the ability to logically detect fabrication errors or circuit malfunctions rather than visually searching for faults with a microscope.

As the complexity of integrated circuits has grown, the ability to logically detect faults using only the normally available inputs and outputs has decreased markedly. As complexity increases, the number of likely faults to be tested for and the number of input vectors required to isolate a specific fault grow rapidly. Unless a design technique is used which allows the tester to examine the interior logic of a chip , the order of magnitude of the number of input vectors required to perform useful logical testing is prohibitive. Thus, if logical testability is desired, a design technique that provides for it must be used.

One such design technique is level sensitive scan design (LSSD) [Ref. 13]. Level sensitive implies that the output of any logic element is dependent only on the levels of its inputs. No logic elements are allowed to depend on a transition such as in an edge triggered flip flop. Scan design implies that all memory elements in the design are to have an auxiliary function where their contents are serially fed to an output pad for examination. This gives a tester the ability to examine intermediate results. In applying the LSSD technique to the adder design, the following steps were taken.

First, all circuits were designed to respond to the level of their inputs and not to require a transition to trigger their operation. Second, to insure that each logic event worked only with stable, non-fluctuating input levels, the inputs to each event were gated. The input gates were

opened only after the inputs were stable (i.e. the outputs of the previous event were stable) and closed before the input gates of the previous event were opened. Third, a dual mode latch was used to store the output of each logic event. In the normal mode of operation, the register latches the outputs of one logic event in parallel and stores them to be used as inputs for the next logic event. In its secondary mode of operation, the register stops taking its parallel inputs and starts to run as a shift register, shifting its contents onto an output pad.

One of the consequences of using the LSSD technique is the large amount of area consumed by the dual mode registers. In high speed operation, an inverter pair would be sufficient to store inter-event results. But to permit low speed testing where the capacitance of a gate may discharge during one clock phase, and provide the dual mode feature, a pair of clocked latches with control circuits is required.

C. LAYOUT DESIGN

With the logic decided upon, the next step was to create the layout of the adder. The logic consisted of four events to produce the sum. Another event was needed to latch the input data onto the chip. A two-phase clock was needed to insure that two adjacent events did not run simultaneously (insuring stable inputs to each event). To make the output of the adder compatible with the input to another adder, a one event delay was added. This insures that the output of one adder does not change while a second adder is using the sum from the first as an input. With two 16-bit addend inputs, one carry-in input, one power supply (Vdd) input, one reference (GND) input, a 16-bit sum output, one carry-out output, and two clock inputs, ten pads were left from a standard 64-pin chip for register mode control input and register (shift mode) output. Since the design called for

five registers, one for each logic event and one for latching the input data, five pads were used for input of the register mode control signals and five were used for the registers to serially output their contents. With the required inputs and output identified, the preliminary floor plan shown in Figure 4.2 was created.

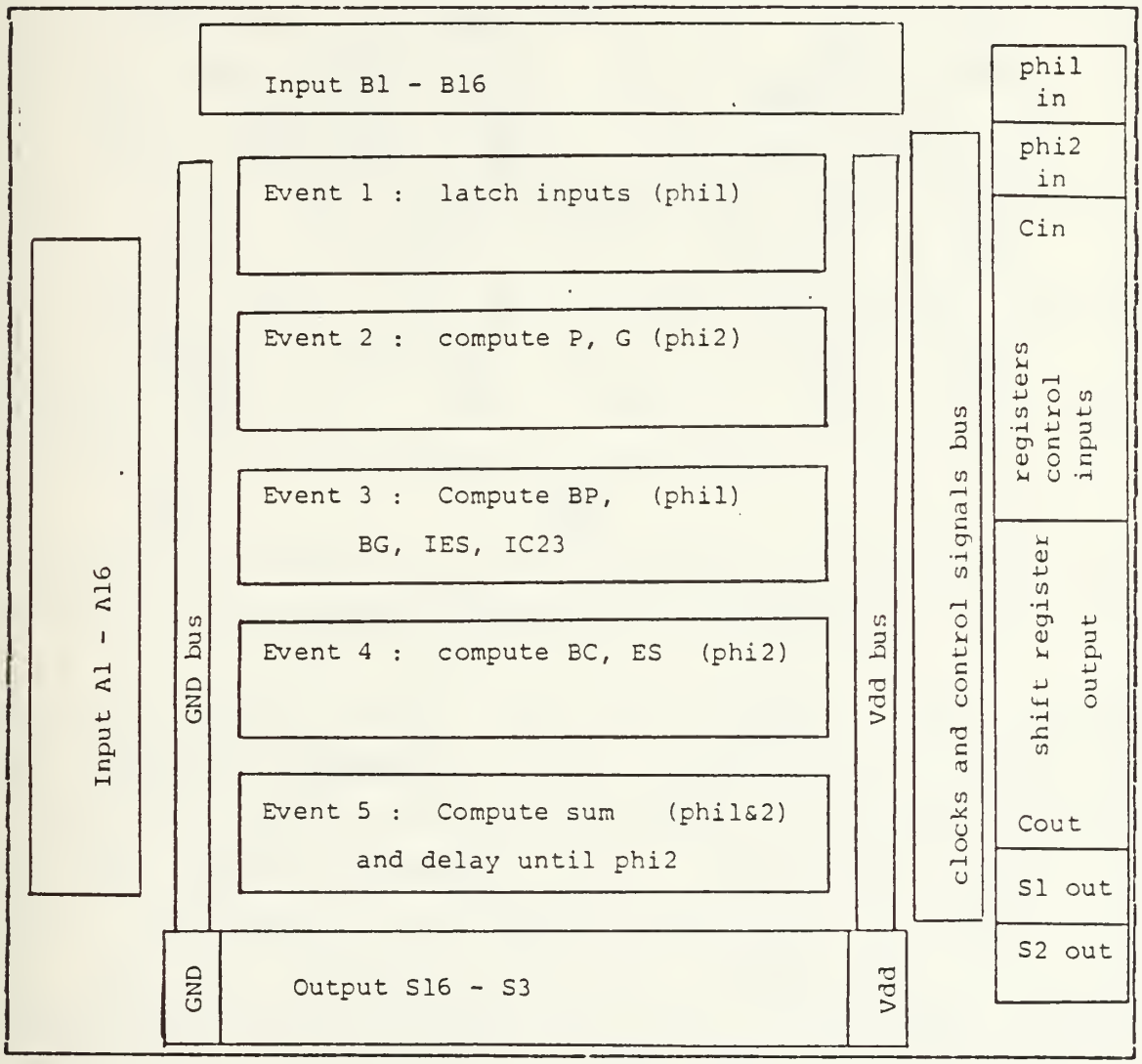


Figure 4.2 Preliminary Chip Floorplan.

The first circuit designed was the dual mode latch of Figure 4.3 Here the circuit is designed to latch the IN level when Control is low ($\overline{\text{Control}}$ is high) and phi1 is high

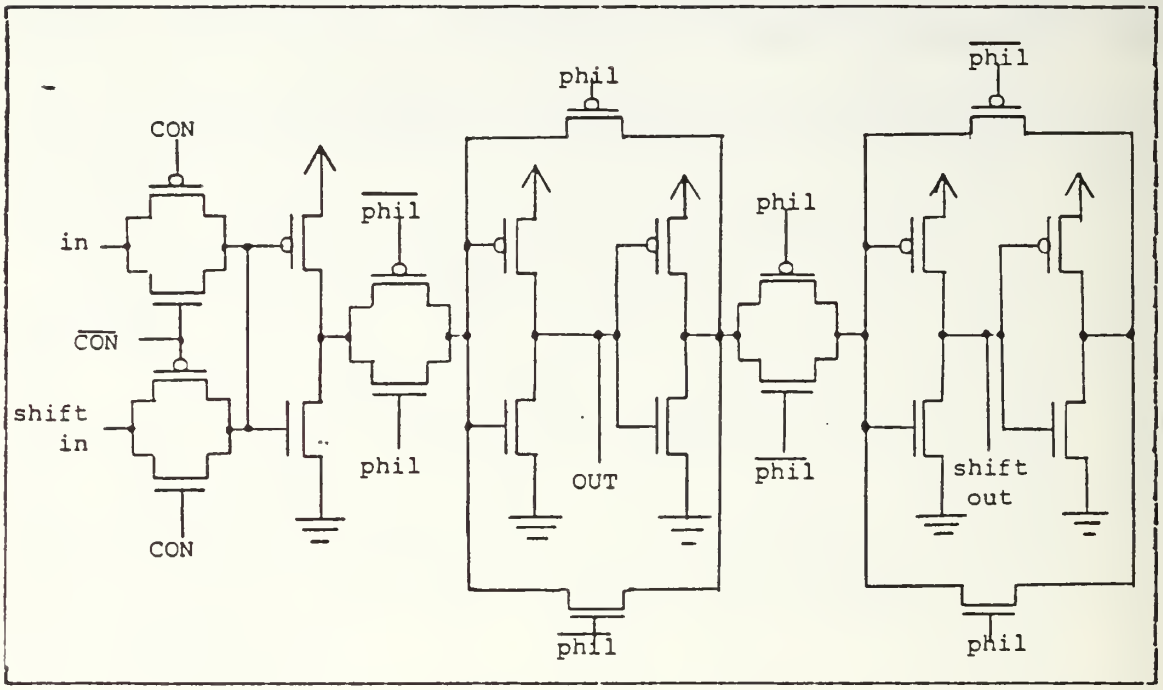


Figure 4.3 Dual Mode Latch.

($\overline{\text{phi1}}$ is low). When phi1 goes low, a copy of the input is also stored in the second latch and becomes available at shift-out which is connected to shift-in of the next latch. When control goes high, the IN signal is blocked and the latch takes its input from the register to the left. The shift-in of the leftmost latch in a register is tied to ground. Versatec plots of the actual layouts of this dual mode latch and the other circuits described in this section are given in Appendix E.

The AND gate used was constructed from a NAND gate followed by an inverter as shown in Figure 4.4 Similarly, the OR gate was constructed from a NOR gate followed by an

inverter (see Figure 4.5). Although logic implemented using these AND and OR gates is more area consuming than the same logic implemented in NAND and NCR gates only, the penalty is not severe because they were used infreguently in the final design.

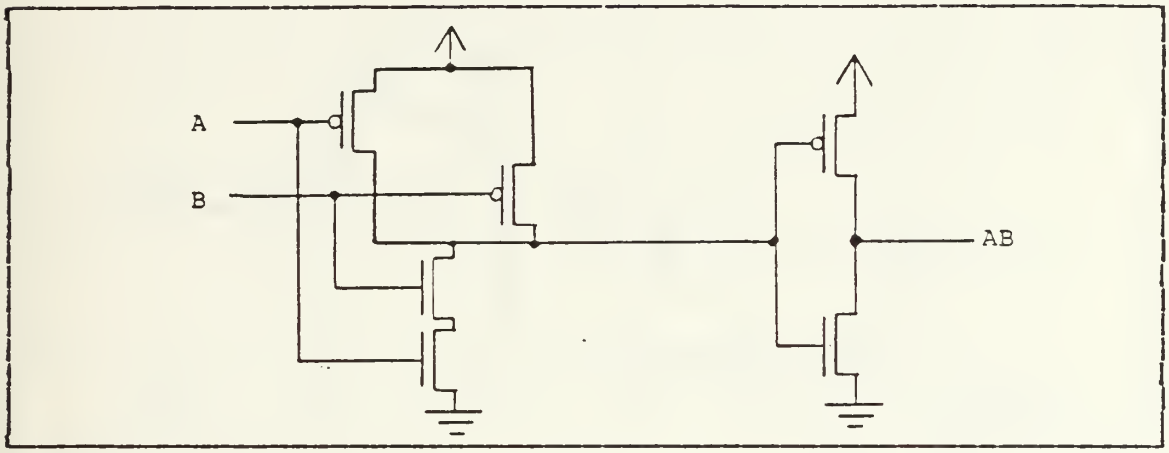


Figure 4.4 AND Gate.

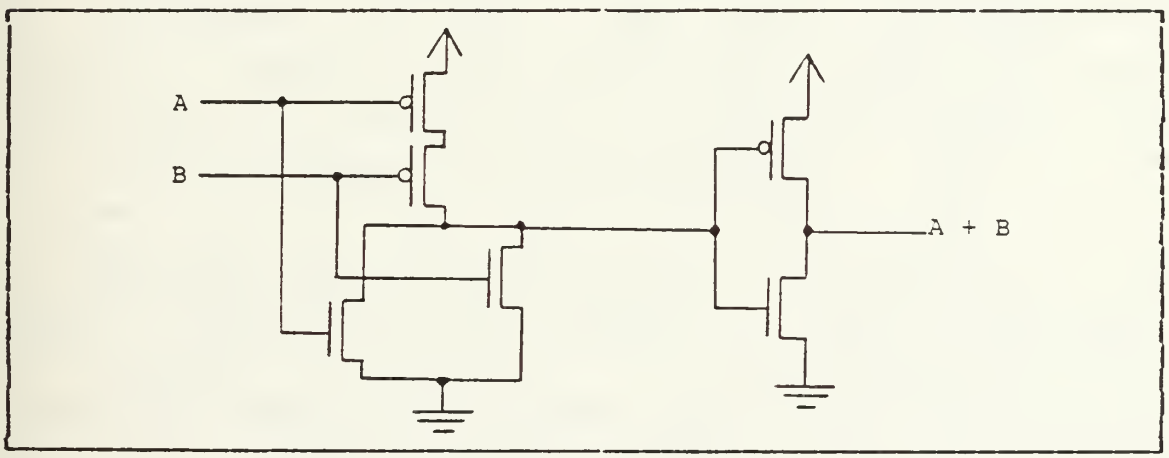


Figure 4.5 OR Gate.

The exclusive OR gate (XOR) was constructed from two inverters and three NAND gates as shown in Figure 4.6 .

Though this design is considerably more area consuming than the XOR gate of Figure 3.1, it was selected because the RNL circuit simulator could correctly model its operation.

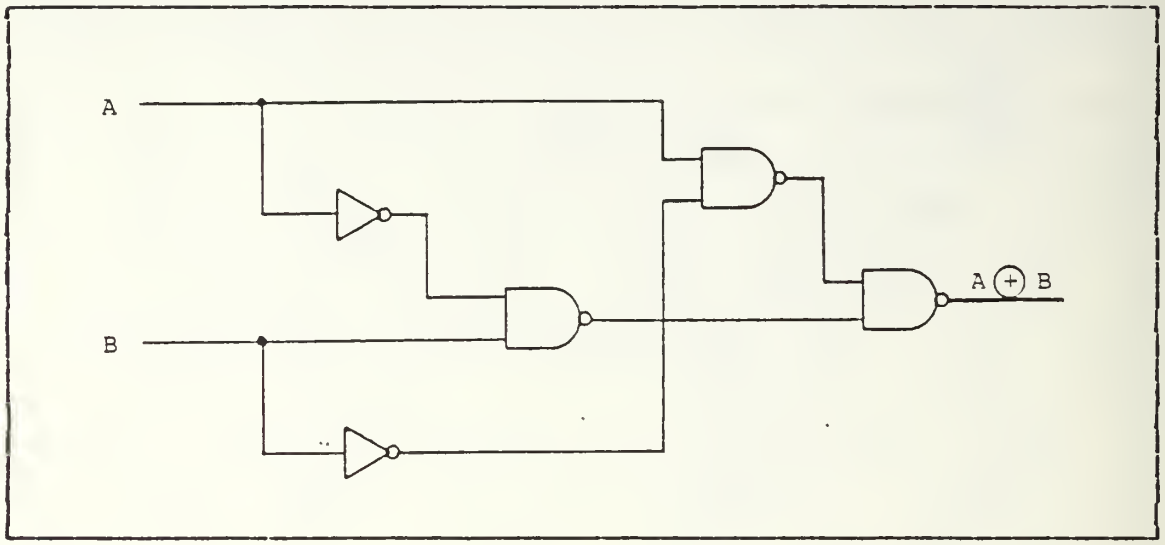


Figure 4.6 Exclusive OR Gate.

More complex logic functions were implemented using programmed logic arrays (PLA) where the outputs are the logical sum (OR) of the products (AND) of inputs. A single phase design was needed. A PLA designed to compute when phi1 is high, between the time the preceding event had produced stable outputs (phi2 going low) and the time phi1 goes low, had to produce the proper sum-of-products results. To hold down fanout, a dynamic structure was needed so that inputs could be applied to a single type of transistor. To prevent steady state power consumption a precharged dynamic structure was needed. Because of charge sharing, the precharging must take place while the inputs are present on the transistor gates of the PLA (see chapter 5, section C, for a complete explanation of the charge sharing problem in this PLA structure). Thus, two distinct events must occur during

this time period. First, the inputs must be applied and precharging must take place. Then evaluation must occur. To cause these two events to occur during a single phase of the clock, the inter-phase time when both phi1 and phi2 are low must be utilized for precharging. The basic structure of the resulting PLA is shown in Figure 4.7

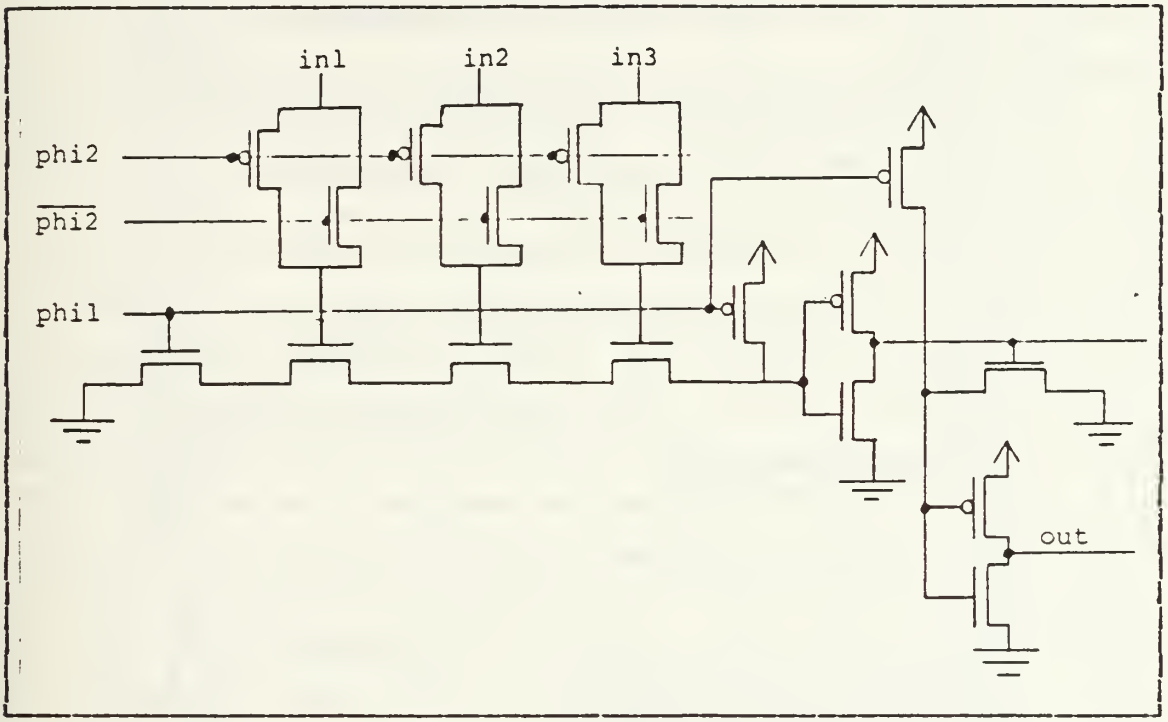


Figure 4.7 PLA Structure.

Referring back to the flocrplan in Figure 4.2, the layout of the circuits which perform the logic of each event are presented in Appendix E. The names assigned to the layouts are given below. Event 1 consists of a 33-bit dual-mode latch. Event 2, which computes the P and G primitives for each bit,is made up of 16 AND gates, 16 XOR gates, and another 33-bit latch. Event 3, which computes the BP and BG primitives, The IES(i)'s and the IC23 for each 4-bit block, is made up of four instances cf PLA82 and a 29-bit latch.

The circuit PLA82 is made up of an 8-input, 5-product, 2-output PLA , two XOR gates, one AND gate, and one OR gate. Event 4, which computes the ES(i)'s and BC for each 4-bit block uses four instances of PLA84 to compute the ES(i)'s and one instance of PLA915 to compute the BC(i)'s and a 21-bit latch. The circuit PLA915 is a 9-input, 15-product, 5-output PLA and the circuit PLA84 is an 8-input, 7-product, 4-output PLA. Event 5 uses four instances of PLA104 to compute the S(i)'s and a 17 bit latch to store results and provide the added delay (by taking the output from the shift out position, the extra clock cycle of delay is generated). The circuit PLA104 is a 10-input, 14-product, 4-output PLA. With this design, the input to output latency is three full cycles of a two-phase non-overlapping clock; three cycles of the clock elapse between the time the addends are presented to the chip and the time the sum becomes available at the output. In the first three registers the odd number of bits is due to the need to store the carry-in value until event 4. In the last two registers the odd number of bits is due to the need to store the computed value of carry-out.

The resulting final layout of Figure 4.8 shows the actual on-chip layout locations of each event's logic. In addition to the logic circuits for each event, the circuits AMP and AMP5 are also seen. These are driver circuits for the high fanout control and clock signals. Each takes as its input a control signal and produces as outputs the control signal and its inverse, both driven by 3-micron x 160-micron transistors. This amplifier is the same design used by the output pads to drive off chip loads.

This final layout represents one implementation of a pipelined CLA adder designed for testability. The relative merits of this design and others that may have been implemented can, as yet, only be qualitatively discussed. The addition of SPICE 2G7 to the CAD toolbag will provide future

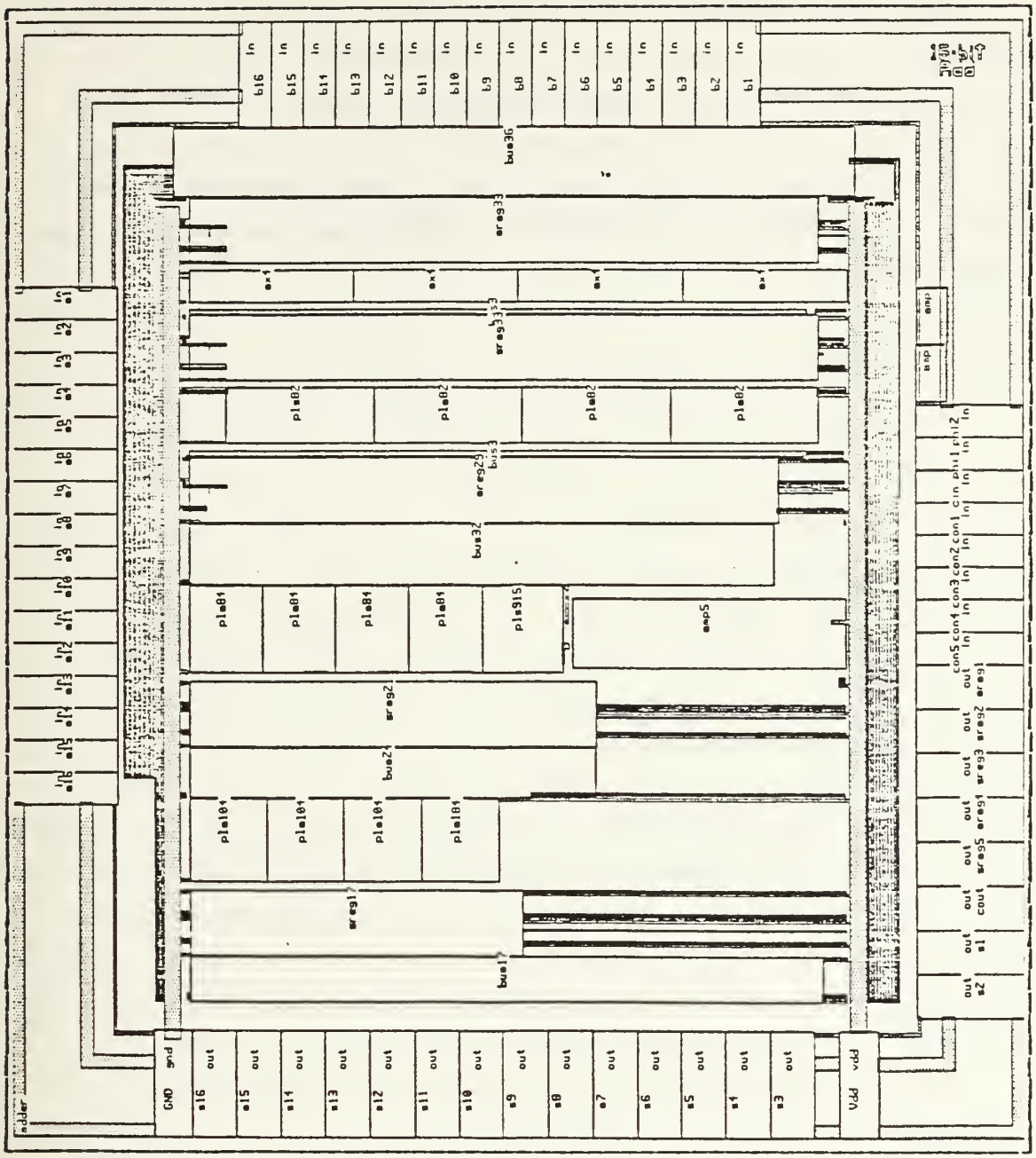


Figure 4.8 Final Layout.

CMOS designers with the quantitative analysis necessary to make decisions involving tradeoffs among primary design objectives.

This final design, when simulated using RNL, functioned properly at clock speeds up to 14 megahertz. Testing of the actual chips produced by MOSIS should give an indication of the accuracy of RNL's predictions. The following chapter presents a test plan to check for proper operation of the adder at low clock rates and tc determine the maximum operating speed.

V. TEST PLAN

After several iterations of the design-simulate-redesign loop, a final layout was achieved for the 16-bit pipelined adder. These iterations provide considerable confidence in the logical correctness of the layout. Appendix D contains RNL simulation results for the full adder. In reading these results it should be kept in mind that the adder requires three cycles of the two-phase clock to produce the sum. In the first part of the simulation, the inputs were kept constant for three clock cycles to facilitate easier reading of the results. With these steady inputs, simulations were run to verify the generation of correct sums, concentrating on those addends that would produce carry propagates and carry generates across the boundaries of the 4-bit blocks. The last part of the simulation utilized different inputs each clock cycle. This was done to test the pipelining feature of the design, insuring no dependence on repeated inputs of the addends to produce the proper sum.

After fabrication of the chip, application of similar inputs to make the same determinations for the actual circuits will form the initial portion of the test plan. In this chapter a test plan for the verification of computational correctness and speed will be presented.

A. INPUTS AND OUTPUTS

The first step in testing the chip will be to connect it to the required input and output circuitry. To accomplish this, the identity of the inputs and outputs on each pin must be determined. Microscopic examination of the chip will reveal the logo "16-bit Add", located between the GND and Vdd buses for the pads in the northeast corner (see

Figure 4.8 which is repeated below for convenience). Using this landmark, the signals on the pads can be labeled as follows.

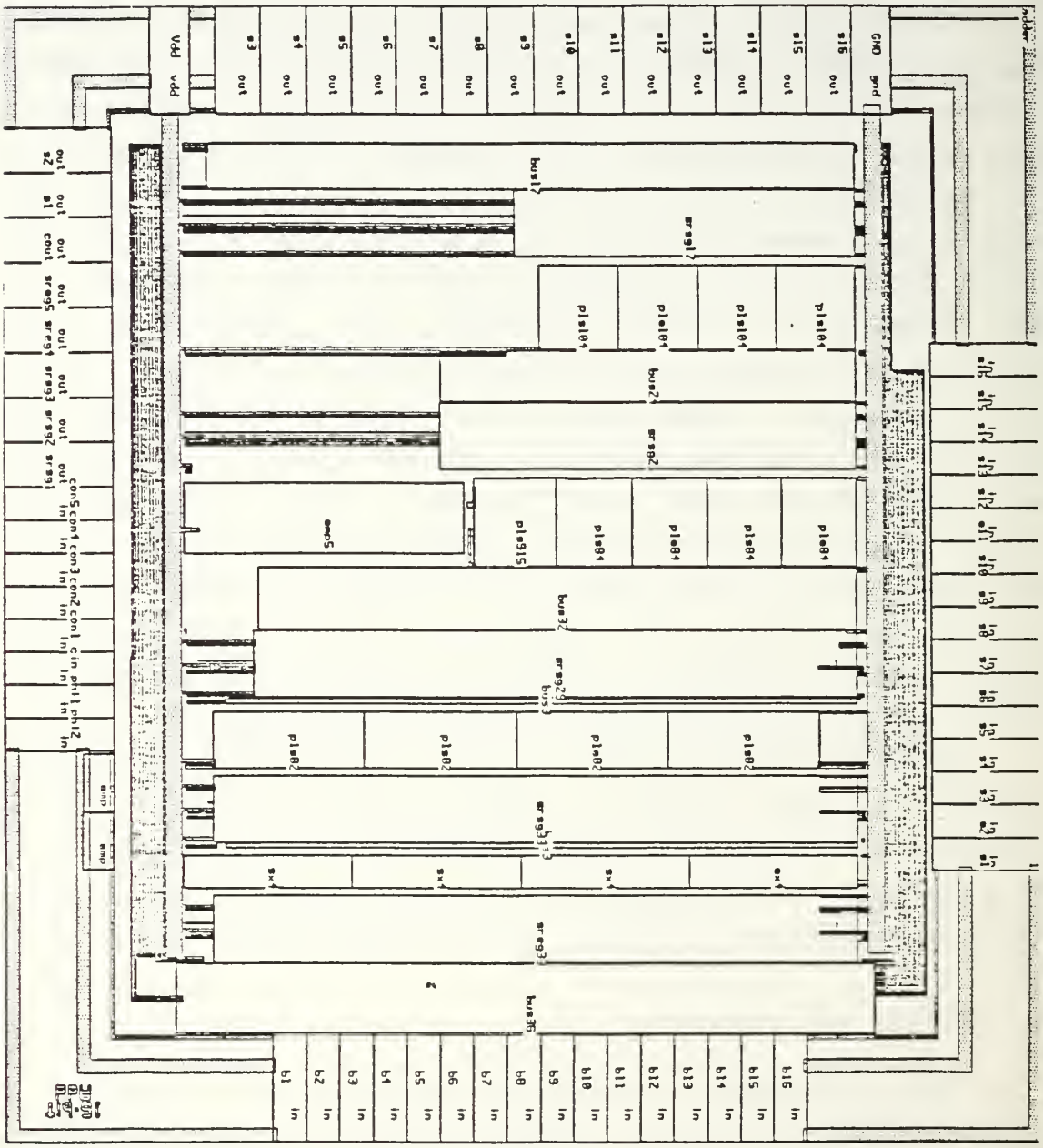


Figure 4.8 (repeated) Final Layout

The western edge has sixteen input pads for the addend A, with the least significant bit, A(1), located at the northern end. The northern edge of the chip also has sixteen input pads for the addend B, with the least significant bit, B(1), located at the eastern end. The southern edge has fourteen output pads and two input pads. At its western end is the GND input pad followed by fourteen output pads for S(16), the most significant bit of the sum, through S(3). Following S(3), at the eastern end is the input pad for Vdd. The eastern edge of the chip has eight input pads and eight output pads. Starting at the northern end, there are input pads for phi1, phi2, Cin, CON1 (control signal for the dual mode register of event 1), CON2, CON3, CON4, and CON5. They are followed by output pads for SREG1 (serial output from dual mode register of event 1), SREG2, SREG3, SREG4, SREG5, Cout, S(2), and S(1) at the southern end.

To supply power to the chip, +5 volts DC should be applied to the Vdd pad and 0 volts to the GND pad. All logical inputs including clocks and control signals should be either GND for a logical 0 or Vdd for a logical 1. Simulation with RNL revealed some restrictions on the clock signals. For proper operation, each clock should remain high for a minimum of 20 nanoseconds and the clock interphase time, when both phi1 and phi2 are low, must be at least 10 nanoseconds in duration. For initial testing, to insure that charge sharing problems caused by too short an interphase time, and fanout problems caused by too short a clock phase duration, are not interpreted as fabrication errors, the clock speed should be adjusted so that both above clock parameters are exceeded by one order of magnitude.

The outputs, like the inputs, are at Vdd to represent a logical 1 and at GND to represent a logical 0. The circuits used to measure the outputs should have high input

impedance, on the order of one megohm. The output pads of the adder are not designed to handle the current source and sink requirements of transistor-transistor logic integrated circuits. The output measurement circuits should be constructed using NMOS or CMOS devicesthat are designed to operate between +5 volts DC and ground.

B. TESTING FOR CORRECT OPERATION

After connecting the adder to a test harness, the next step is to verify the generation of correct sums by the adder. There are several inputs that should be included in the testing to verify the correct operation of individual circuits. These are contained in Appendix F. In addition to the test vectors of Appendix F, several randomly selected input vectors should be tested. If the adder should fail to generate correct sums, The LSSD features can be employed to examine intermediate results.

1. Intermediate results

With the LSSD design, a tester can leave input levels constant for a long period of time and use the shift mode of the internal registers to examine the internal state of the chip. The rightmost bit of each register is always available at the output pad for that register. To obtain the contents of the other bits, the control signal for the given register is set to and held at logical 1 while the clock continues to run. For registers 1, 3, and 5 the serial output will be meaningful and stable while phi2 is high. The serial output of registers 2 and 4 will be stable when phi1 is high. Table 3 lists in order the intermediate values available at the SREG(n) output pad when the input CONn is high.

TABLE 3

Register Serial Outputs

Clock Cycle	SREG1	SREG2	SREG3	SREG4	SREG5
0	B1	P1	BP1	Cin	S1
1	B2	P2	IES3	BC2	S3
2	B3	P3	IES4	Cout	S5
3	B4	P4	BG2	ES2	S7
4	B5	P5	IES5	ES4	S9
5	B6	P6	IES6	ES6	S11
6	B7	P7	IC67	ES8	S13
7	B8	P8	BP3	ES10	S15
8	B9	P9	IES11	ES12	0
9	B10	P10	IES12	ES14	Cout
10	B11	P12	BG4	ES16	S2
11	B12	P12	IES13	BC1	S4
12	B13	P13	IES14	BC3	S6
13	B14	P14	IC1415	ES1	S8
14	B15	P15	BG1	ES3	S10
15	B16	P16	IES1	ES5	S12
16	A1	G1	IES2	ES7	S14
17	A2	G2	IC23	ES9	S16
18	A3	G3	BP2	ES11	0
19	A4	G4	IES7	ES13	0
20	A5	G5	IES8	ES15	0
21	A6	G6	BG3	0	0
22	A7	G7	IES9	0	0
23	A8	G8	IES10	0	0
24	A9	G9	IC1011	0	0
25	A10	G10	BP4	0	0
26	A11	G11	IES15	0	0
27	A12	G12	IES16	0	0
28	A13	G13	Cin	0	0
29	A14	G14	0	0	0
30	A15	G15	0	0	0
31	A16	G16	0	0	0
32	Cin	Cin	0	0	0
33	0	0	0	0	0
34	0	0	0	0	0

C. TESTING FOR SPEED OF OPERATION

Once the chips containing fabrication errors have been culled from the chip set returned by MOSIS, the task remaining is to determine just how fast the adder can run. Rather than simply increasing the clock rate until the adder fails, the duration of the time both phi1 and phi2 are high, and the interphase time should reduced separately. RNL simulation indicates that the circuit which generates S4 within PLA104 is the limiting circuit for clock phase duration (i.e. it requires the longest time to correctly

evaluate its inputs). RNL simulation also indicates that the circuits in PLA104 which generate S1 and S4 are the limiting circuits for the clock interphase duration.

Since the PLA is constructed of precharged dynamic circuits, the evaluation clock phase must be long enough to allow the inputs to drive the outputs to their proper values, even if the inputs are the same as those of the previous evaluation cycle. This allows the tester to use a constant input as the duration of each clock phase is reduced until the adder produces incorrect results.

Determination of the clock interphase duration limit is more difficult. This is because the inputs to a PLA must be changing to cause charge sharing problems to occur. For

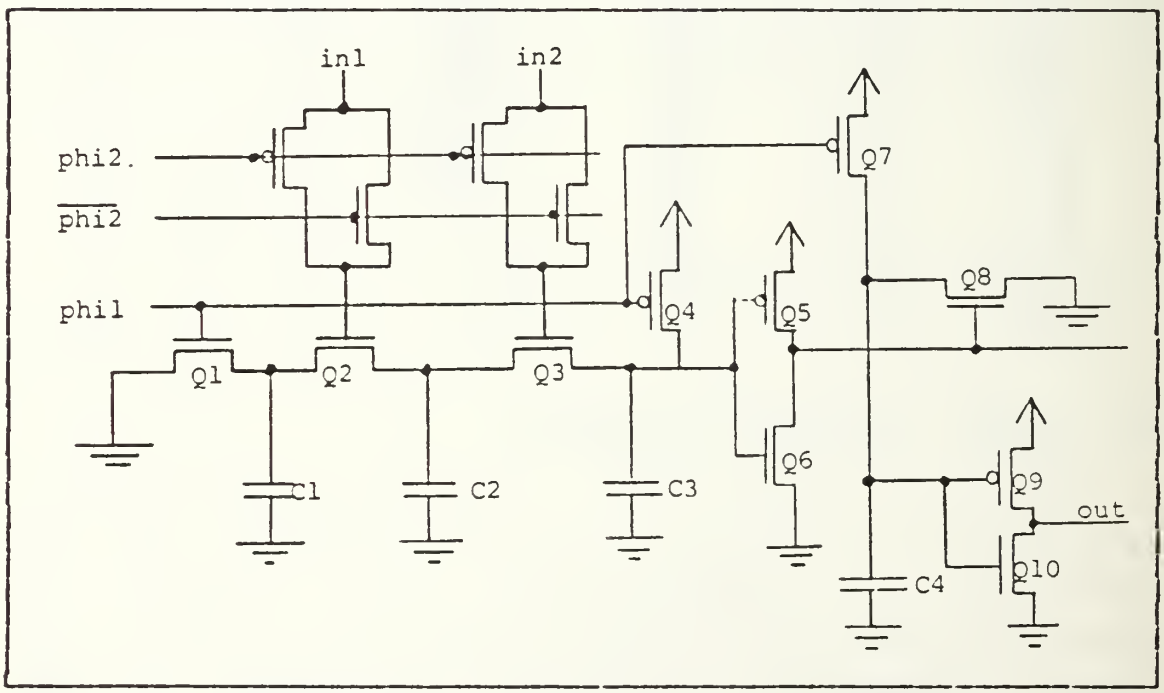


Figure 5.1 Charge Sharing in a PLA.

example, in Figure 5.1 assume that the first set of inputs is in1=1, in2=0, and that this is correctly evaluated to

produce out=0 when phi1 is high. Now assume that the next input is in1=0 and in2=1, which should also evaluate to out=0. However, if the precharge time (when the inputs are present on the gates of Q2 and Q3 and phi1 is still low) is insufficient, C2 will not be charged to Vdd when precharging ends (C2 was discharged to zero volts during the previous evaluation when in1 was high and phi1 was high). Now, when evaluation begins (phi1 going high) the low voltage across C2 causes Q5 and Q6 to interpret their input as a logical 0. As a result the output of the Q5-Q6 inverter pair goes high, causing Q8 to turn on, discharging C4 and resulting in an output of logical 1, which is incorrect. Table 4 lists the proper evaluation sequence when precharge time is sufficient and the improper sequence due to insufficient precharge time. In this table, for the inputs, output, and capacitor voltages a 1 indicates Vdd, 0 indicates GND, and X indicates somewhere in between. For the transistors, a 1 indicates on, a 0 indicates off, and an X indicates neither fully on

TABLE 4

PLA Evaluation Sequences

Proper evaluation sequence:

phi 1 2	in 12	C 1234	Q 1234567890 1	out
1 0	10	0011	1100010C01	0
0 0	10	0011	0101011C01	0
0 1	01	0011	0101011001	0
0 0	01	0111	0011011C01	0
1 0	01	0111	1010010C01	0

Improper evaluation sequence:

phi 1 2	in 12	C 1234	Q 1234567890 1	out
1 0	10	0011	1100010C01	0
0 0	10	0011	0101011C01	0
0 1	01	0011	0101011C01	0
0 0	01	0X11	0011011C01	0
1 0	01	0XX0	1010XX0X10	1

nor fully off. Subsequent inputs of in1=0 and in2=1 may produce correct results since with constant inputs, each precharge time will add more charge to C2 until there is sufficient charge to allow the output of the Q5-Q6 inverter to remain low.

Thus, to check for charge sharing problems in the circuit of Figure 5.1, the inputs must alternate. Likewise, in PLA104 to check for charge sharing errors in output S1, its inputs must alternate between ES1=0, BC=0 and ES1=1, BC=1 as the interphase time is reduced. This can be accomplished for all four instances of PLA104 simultaneously by alternating inputs of

A = 0001 1001 1001 1001
B = 0000 1000 1000 1000
Cin = 1

and

A = 0000 0000 0000 0000
B = 0000 0000 0000 0000
Cin = 0

To check for charge sharing errors in S4, the inputs to PLA 104 must cycle between BC=1, S4=0, S3=S2=1,S1=0 and BC=0,S4=0,S3=S2=S1=1. This may be accomplished for all four instances of PLA104 simultaneously by alternating inputs of

A = 0110 1110 1110 1110
B = 0000 1000 1000 1000
Cin = 1

and

A = 0111 0111 0111 0111
B = 0000 0000 0000 0000
Cin = 0

This maximum speed testing assumes that RNL has correctly identified the slowest circuits on the chip. RNL

simulations have indicated that the next slowest circuit (PLA915) is at least 20% faster than PLA104 (16.0 nsec for PLA915 vs. 20.1 nsec for PLA104). Also, all other PLA's functioned properly with a 5 nsec interphase time.

Should PLA104 prove to be the speed limiting circuit for the chip, the actual failure speeds of the chip can serve as an indication of the accuracy of the RNL simulation for future designs.

VI. CONCLUSIONS

The experience gained in the design of the adder coupled with the clarity of hindsight leads to the following conclusions and recommendations.

A. THE CMOS TECHNOLOGIES

The CMOS technologies will play a role of steadily increasing importance in the VLSI designs of the future. MOSIS is already offering, on an experimental basis, CMOS Bulk p-well fabrication with a one-micron minimum feature size. A scalable set of design rules, to allow initial fabrication in 3-micron CMOS for design verification before the far more expensive 1-micron process is used, is being developed.

In the private sector there is considerable research aimed at finding an insulating substrate material that does not have the variability and thermal problems of sapphire. Progress in this area will remove the drawback caused by latchup tendencies in CMOS Bulk.

B. CMOS CAD TOOLS

Though the design tools currently available at NPS constitute a complete set for the design of CMOS Bulk p-well circuits, the recent CAD tool set released by the University of Washington/Northwest VLSI Consortium, Release 2.0 [Ref. 11], coupled with University of California at Berkeley Winter 1983 CAD tools, represents a more complete and cohesive set for CMOS design. When sufficient disk space on the Vax 11-780 becomes available to load the Release 2.0, implementation of the Release 2.0 CAD package

is highly recommended. An added benefit of installing the Release 2.0 package is the cell library provided. The library contains several basic standard cells with known performance characteristics. The library also contains the standard pad frames used by MOSIS. Though MOSIS does not require the use of standard pad frames on designs submitted, their use does speed up fabrication.

As mentioned earlier, as soon as SPICE 2G7 is available, its addition to the CAD toolbag would be most advantageous to a CMOS designer.

C. DESIGN OF THE ADDER

If the design of the adder were to be undertaken again, a different approach to generating the sum would probably have been used, especially if the new CAD tools mentioned above were available. The logic approach to the computation would still involve CLA addition, but it would be accomplished using combinational logic and library cells rather than PLA's. Testability would probably suffer greatly, but effort would be made to reduce the sum generation to two logical events. Though the level of testability provided by the current design should provide considerable insight into CMOS Bulk p-well performance and CAD tool accuracy, there would be no need to repeat the investigation.

APPENDIX A

SPICE MODEL CARDS FOR 3-MICRON CMOS-PW DEVICES

CMU models for MOSIS 3-micron CMOS Bulk p-well devices:

Fast Models

```
.model  n  nmos  vto=0.4  tox=0.7e-7  lambda=1e-7  ld=1e-6
+xj=1.1e-6  gamma=.3  uo=500  cbd=5e-4  cbs=5e-4

.model  p  pmos  vto=-.4  tox=0.7e-7  lambda=1e-7  ld=1e-6
+xj=1.1e-6  gamma=.3  uo=300  cbd=3.5e-4 cbs=3.5e-4
```

Slow Models

```
.model  n  nmos  vto=1.0  tox=0.8e-7  lambda=1e-7  ld=.5e-6
+xj=0.6e-6  gamma=1.3 uo=400  cbd=6e-4  cbs=6e-4
.model  p  pmos  vto=-1.0 tox=0.8e-7  lambda=1e-7  ld=.5e-6
+xj=0.6e-6  gamma=.9  uo=200  cbd=4.1e-4  cbs=4.1e-4
```

MIT Models for MOSIS 3-micron CMOS Bulk p-well devices:

Slow - Slow

```
.model nss nmos  level=2  rsh=30  tox=650e-10  ld=.25e-6
+xj=.35e-6  cj=6e-4  cjsw=4e-10  wo=475  vto=1.2
+cgso=1.3e-10  cgdo=1.3e-10  nsub=1.5e16
+vmax=5e4  pb=.7  mj=.5  mjsw=.5
+neff=2.5  ucrit=8e4  uexp=.25

.model pss pmos  level=2  rsh=80  tox=650e-10  ld=.25e-6
+xj=.35e-6  cj=4.1e-4 cjsw=2.5e-10  uo=190  vto=-1.2
+cgso=1.3e-10  cgdo=1.3e-10  nsub=5e15  tpg=-1
+vmax=5e4  pb=.7  mj=.5  mjsw=.5
+neff=2.5  ucrit=8e4  uexp=.15
```

```
    Fast p-type  Slow n-type

.model nfs nmos  level=2  rsh=30  tox=600e-10  ld=.25e-6
+xj=.35e-6  cj=6.0e-4 cjsw=4.0e-10  uo=475  vto=1.2
+cgso=1.9e-10  cgdo=1.9e-10  nsub=1.5e16
+vmax=5e4  pb=.7  mj=.5  mjsw=.5
+neff=2.5  ucrit=8e4  uexp=.25

.model pfs pmos  level=2  rsh=20  tox=600e-10  ld=.40e-6
+xj=.60e-6  cj=2.0e-4 cjsw=1.0e-10  uo=270  vto=-0.6
+cgso=2.0e-10  cgdo=2.0e-10  nsub=0.3e15  tpg=-1
+vmax=5e4  pb=.7  mj=.5  mjsw=.5
+neff=2.0  ucrit=8e4  uexp=.15


    Fast p-type  Fast n-type

.model nff nmos  level=2  rsh=10  tox=550e-10  ld=.40e-6
+xj=.60e-6  cj=3.0e-4 cjsw=2.0e-10  uo=675  vto=0.6
+cgso=2.5e-10  cgdo=2.5e-10  nsub=0.5e16
+vmax=5e4  pb=.7  mj=.5  mjsw=.5
+neff=2.5  ucrit=8e4  uexp=.25

.model pff pmos  level=2  rsh=20  tox=550e-10  ld=.40e-6
+xj=.60e-6  cj=2.0e-4 cjsw=1.0e-10  uo=270  vto=-0.6
+cgso=2.5e-10  cgdo=2.5e-10  nsub=0.3e15  tpg=-1
+vmax=5e4  pb=.7  mj=.5  mjsw=.5
+neff=2.0  ucrit=8e4  uexp=.15
```

```
    Slow p-type    Fast n-type

.model nsf nmos  level=2  rsh=10  tox=600e-10  ld=.40e-6
+xj=.60e-6  cj=3.0e-4 cjsw=2.0e-10  uo=675  vto=0.6
+cgso=2.0e-10  cgdo=2.0e-10  nsub=0.5e16
+vmax=5e4   pb=.7   mj=.5   mjsw=.5
+neff=2.5   ucrit=8e4   uexp=.25

.model psf pmos  level=2  rsh=80  tox=600e-10  ld=..25-6
+xj=.35e-6  cj=4.1e-4 cjsw=2.5e-10  uo=190  vto=-1.2
+cgso=1.2e-10  cgdo=1.2e-10  nsub=5.0e15   tpg=-1
+vmax=5e4   pb=.7   mj=.5   mjsw=.5
+neff=2.0   ucrit=8e4   uexp=.15
```

APPENDIX B

UNIX MANUAL ENTRY FOR RULEC

NAME

rulec – Compile design rules for Lyra

SYNOPSIS

rulec [–lo] rules

DESCRIPTION

Rulec is a shell script with the following processing steps:

i) The actual *Lyra* rule compiler is invoked to translate the symbolic rule description, *rules*.**r**, to lisp code, rules.**l**

ii) The lisp compiler, *Liszt*, is invoked to compile *rules*.l to *rules*.**o**

iii) *rules*.**o** is loaded into *Lyra.proto* to generate an executable lisp *Lyra*, *rules*.

iv) The intermediate files *rules*.l, and *rules*.**o** are deleted.

The following options are supported:

–l **(load only)** No compilation is done. Previously compiled rules, *rules*.o, are loaded into *Lyra.proto* to generate an executable Lyra, *rules*. This option is useful mainly at Berkeley, where *Lyra.proto* changes frequently.

–o (save object) *Name*.**o** *is not removed. Enables* '**rulec** –**l** *rules*' in the future.

FILES

~cad/bin/rulec – rulec shell script.
~cad/lib/lyra/Rulec1 – lisp rule compiler
~cad/lib/lyra/Lyra.proto – Lyra sans compiled rules code.
~cad/lib/lyra/*.r -- standard rulesets.
~cad/lib/lyra/DEFAULTS -- gives default rulesets for Caesar technologies.

SEE ALSO

Lyra (CAD)
Liszt (1)

AUTHOR

Michael Arnold.

APPENDIX C
PRESIM USER'S GUIDE

Config file: used to calibrate RNL

```
capm2a      .00000
capm2p      .00000
capma       .00006
capmp       .00000
cappa       .00006
cappp       .00000
capda       .00010
capdp       .00060
cappda      .00010
cappdp      .00060
capga       .00057
lambda      1.0
```

PRESIM User's Guide

UW/NW VLSI Consortium

Department of Computer Science
University of Washington
Seattle, WA 98195

(This document is based on portions of the document "User's Guide to NET, PRESIM and RNL/NL," by Christopher J. Terman, Laboratory for Computer Science, M.I.T., Cambridge, MA 02139.)

One must first convert the .sim file to a network file suitable for use by RNL or NL – to do this we run PRESIM:

presim foo.sim foo [config] options...

which converts the file foo.sim into a binary file for RNL/NL called foo.

The -g option:

> Suppresses the sum-of-products formation. This may be desired if you think sum-of-products is formed wrong otherwise the advantages of the transistor and node reduction make this option unattractive.

The -c option:

> -cfile,minvalue

writes a list of node names and capacitances to the specified file. Only capacitances larger than minvalue will be included.

The -t option:

> -tfile,minvalue

writes a list of transistors and RC values to the specified file – there are two entries for each transistor. The R's come from the size of the transistor, C's from the source/drain capacitance. Only RC values larger than minvalue will be included.

The -p option:

> -presist,voltage

provides a worse-case estimate of the circuit power consumption by assuming that all the pullups (DEP or LOWP devices with drain=VDD) are all on simultaneously. "Voltage" specifies the supply

voltage, for example "-p5" specifies a VDD of 5 volts. The result is printed after PRESIM completes its other processing. When figuring the resistance of a pullup device the "power" characteristic resistance as set in the config file is used.

The optional third file (config) specifies various electrical parameters. The internal values (the defaults) are a generic set. They do not reflect any particular fabrication process. (UW-NW VLSI NOTE: A configuration file is provided in the source code that duplicates the internal settings as an example of how this file could be used. In addition we note that, the resistor values are stored first sorted by width, then by length not by the ratio. Values not explicitly provided in the configuration file are estimated by linear interpolation.) The format of this file is lines of the form

parameter value comments...

Lines beginning with ";" are treated as all comment. The parameter names and their default values are:

```
; configuration file for "standard" MPC process

capm2a .00000  ; 2nd metal capacitance -- area, pf/sq-micron
capm2p .00000  ; 2nd metal capacitance -- perimeter, pf/micron
capma  .00003  ; 1st metal capacitance -- area, pf/sq-micron
capmp  .00000  ; 1st metal capacitance -- perimeter, pf/micron
cappa  .00004  ; poly capacitance -- area, pf/sq-micron
cappp  .00000  ; poly capacitance -- perimeter, pf/micron
capda  .00010  ; n-diffusion capacitance -- area, pf/sq-micron
capdp  .00060  ; n-diffusion capacitance -- perimeter, pf/micron
cappda .00010  ; p-diffusion capacitance -- area, pf/sq-micron
cappdp .00060  ; p-diffusion capacitance -- perimeter, pf/micron
capga  .00040  ; gate capacitance -- area, pf/sq-micron

lambda 2.5     ; microns/lambda (conversion from .sim file units
               ; to units used in cap parameters)

lowthresh  0.3 ; logic low threshold as a normalized voltage
highthresh 0.8 ; logic high threshold as a normalized voltage

cntpullup 0    ; <>0 means that the capacitor formed by gate of
               ; pullup should be included in capacitance of output
               ; node

diffperim 0    ; <>0 means do not include diffusion perimeters
               ; that border on transistor gates when figuring
               ; sidewall capacitance (*)

subparea 0     ; <>0 means that poly over transistor region will not
               ; be counted as part of the poly-bulk capacitor (*)
```

```
diffext 0       ; diffusion extension for each transistor, i.e., each
                ; transistor is assumed to have a rectangular source
                ; and drain diffusion extending diffext units wide and
                ; transistor-width units high.  The effect of the
                ; diffusion extension is to add some capacitance to
                ; the source and drain node of each transistor --
                ; useful when processing the output of NET to improve
                ; the capacitive loading approximations without adding
                ; explicit load capacitors.  diffext is specified in
                ; lambda (it will be converted using the lambda factor
                ; above).

resistance channel context width length resist
; this command specifies the equivalent resistance for a transistor
; of type channel with the specified width and length.  Transistors
; matching this entry will have the specified resistance; linear
; interpolation is done if the width and/or length is not matched
; exactly.
;  channel is one of "enh", "dep", "intrinsic", "low-power",
;                    "pullup", or "p-chan"
;  context is one of "static", "dynamic-high", "dynamic-low", or "power"
;  width is given in lambda
;  length is given in lambda
;  resist is given in ohms
```

(*) These paramters should be 1 only when processing the output of the node extractor. They cause various corrections to be made to the interconnect component of a node's capacitance -- usually only extracted .sim files have information regarding interconnect capacitance.

PRESIM uses these parameters in calculating the capacitance for each electrical node and the resistance for each transistor channel.

APPENDIX D
ADDER SIMULATION

The following two listings are: (1) the RNL command file for the entire chip and (2) the results of running that command file. In addition to this overall testing, all the layout of Appendix G were simulated individually. A nice feature of RNL is the indication of when a watched node changes state. Thus, by making all the outputs of a circuit watched nodes, RNL will provide the minimum time duration for a clock cycle to produce the outputs (the longest time indicated by the simulation). This can be confirmed by running the simulation with a faster clock, resulting in outputs of X (neither 1 nor 0) where insufficient time has been allowed.

RNL simulation to determine the minimum time for precharging the PLA circuits is only slightly more involved. For each product term in the PLA, alternating inputs are selected that will result in maximum amount of N+ diffusion needing to be charged from 0 volts to Vdd. Then as these inputs are alternated, the PLA precharge time is reduce until the circuit fails to produce correct results. For the PLA's in the adder, visual inspection for the product term with the longest precharge requirement was done by looking for the longest N+ diffusion line which must be charged through the maximum number of transistors. The visual inspection results were confirmed by RNL simulations.

```
(loo-file "chip.loo")
(load "uwstd.l")
(load "uwsim.l")
(read-network "chip")
(setc nodes '(a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11 a12 a13
a14 a15 a16 b1 b2 b3 b4 b5 b6 b7 b8 b9 b10 b11 b12 b13 b14 b15
b16 s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13 s14 s15 s16
cin cout phi1 phi2 con1 con2 con3 con4 con5))
(chflag nodes)
1 a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11 a12 a13 a14 a15 a16
1 b1 b2 b3 b4 b5 b6 b7 b8 b9 b10 b11 b12 b13 b14 b15 b16
1 con1 con2 con3 con4 con5
1 cin phi1 phi2
(defvec '(bin clocks phi1 phi2))
(defvec '(bin aaaa a15 a15 a14 a13 a12 a11 a10
                a9 a8 a7 a6 a5 a4 a3 a2 a1))
(defvec '(bin bbbb b16 b15 b14 b13 b12 b11 b10
                b9 b8 b7 b6 b5 b4 b3 b2 b1))
(defvec '(bin sum cout s16 s15 s14 s13 s12 s11 s10
                s9 s8 s7 s6 s5 s4 s3 s2 s1))
(def-report '("state is now:" (vec clocks) cin cout newline
              (vec aaaa) newline
              (vec bbbb) newline
              (vec sum)))
(defun ss(dummy)
    (step incr))
(defun cycles (a)
    (repeat i 1 a
        (setc incr 100)
        (ss '(x))
        (seto incr 256)
        (h '(phi1))
        (ss '(x))
        (setc incr 100)
        (l '(phi1))
        (ss '(x))
        (seto incr 256)
        (h '(phi2))
        (s '(x))
        (l '(phi2))
    )
)
(cycles 5)
(invec '(aaaa 0b0000111100001111))
(invec '(bbbb 0b1111000011110000))
(cycles 3)
(invec '(bbbb 0b0000000100000001))
(cycles 3)
h cin
(cycles 3)
l cin
(invec '(aaaa 0b1111111111111111))
(invec '(bbbb 0b0000000000000000))
(cycles 3)
h cin
(cycles 3)
```

```
l cin
(invec '(bbbb 0b0000000000000001))
(cycles 3)
(invec '(bbbb 0b0000000000000000))
(cycles 1)
(invec '(aaaa 0b0000000000000000))
(cycles 1)
(invec '(bbbb 0b1111111111111111))
(cycles 1)
(invec '(aaaa 0b0000100000000000))
(cycles 1)
(invec '(bbbb 0b1111011111111111))
(cycles 1)
h cin
(cycles 1)
(invec '(aaaa 0b0000000000000000))
(invec '(bbbb 0b0000000010000000))
(cycles 1)
l cin
(cycles 4)
```

```
Dec  6 15:23 1984  chip.log Page 1


Loading uwsim.l
Done loading uwsim.l
; 3086 nodes, transistors: enh=1494 intrinsic=0 p-chan=1141 dep=0 low-power=0 pullup=0 resisto

Step begins @ 0 ns.
ph12=0 @ 0
ph11=0 @ 0
cin=0 @ 0
con5=0 @ 0
con4=0 @ 0
con3=0 @ 0
con2=0 @ 0
con1=0 @ 0
b16=0 @ 0
b15=0 @ 0
b14=0 @ 0
b13=0 @ 0
b12=0 @ 0
b11=0 @ 0
b10=0 @ 0
b9=0 @ 0
b8=0 @ 0
b7=0 @ 0
b6=0 @ 0
b5=0 @ 0
b4=0 @ 0
b3=0 @ 0
b2=0 @ 0
b1=0 @ 0
a16=0 @ 0
a15=0 @ 0
a14=0 @ 0
a13=0 @ 0
a12=0 @ 0
a11=0 @ 0
a10=0 @ 0
a9=0 @ 0
a8=0 @ 0
a7=0 @ 0
a6=0 @ 0
a5=0 @ 0
a4=0 @ 0
a3=0 @ 0
a2=0 @ 0
a1=0 @ 0

Step begins @ 10 ns.
ph11=1 @ 0

Step begins @ 35 ns.
ph11=0 @ 0

Step begins @ 45 ns.
ph12=1 @ 0
s16=0 @ 14.2
s9=0 @ 16.4
```

```
s11=0 @ 16.4
s13=0 @ 16.4
s15=0 @ 16.4
s7=0 @ 16.4
s5=0 @ 16.4
s3=0 @ 16.4
s14=0 @ 16.5
s12=0 @ 16.5
s10=0 @ 16.5
s8=0 @ 16.5
s6=0 @ 16.5
s4=0 @ 16.5
s2=0 @ 16.7
s1=0 @ 20
state is now:
Current time= 70
clocks=0b01 cin=0 cout=X
aaaa=0b0000000000000000
bbbb=0b0000000000000000
sum=X0000000000000000

Step begins @ 70 ns.
phi2=0 @ 0

Step begins @ 80 ns.
phi1=1 @ 0

Step begins @ 105 ns.
phi1=0 @ 0

Step begins @ 115 ns.
phi2=1 @ 0
cout=0 @ 22.9
state is now:
Current time= 140
clocks=0b01 cin=0 cout=0
aaaa=0b0000000000000000
bbbb=0b0000000000000000
sum=0b0000000000000000

Step begins @ 140 ns.
phi2=0 @ 0

Step begins @ 150 ns.
phi1=1 @ 0

Step begins @ 175 ns.
phi1=0 @ 0

Step begins @ 185 ns.
phi2=1 @ 0
state is now:
Current time= 210
clocks=0b01 cin=0 cout=0
aaaa=0b0000000000000000
bbbb=0b0000000000000000
```

```
sum=0b0000000000000000

Step begins @ 210 ns.
phi2=0 @ 0

Step begins @ 220 ns.
phi1=1 @ 0

Step begins @ 245 ns.
phi1=0 @ 0

Step begins @ 255 ns.
phi2=1 @ 0
state is now:
Current time= 280
clocks=0b01 cin=0 cout=0
aaaa=0b0000000000000000
bbbb=0b0000000000000000
sum=0b0000000000000000

Step begins @ 280 ns.
phi2=0 @ 0

Step begins @ 290 ns.
phi1=1 @ 0

Step begins @ 315 ns.
phi1=0 @ 0

Step begins @ 325 ns.
phi2=1 @ 0
state is now:
Current time= 350
clocks=0b01 cin=0 cout=0
aaaa=0b0000000000000000
bbbb=0b0000000000000000
sum=0b0000000000000000

Step begins @ 350 ns.
b16=1 @ 0
b15=1 @ 0
b14=1 @ 0
b13=1 @ 0
b8=1 @ 0
b7=1 @ 0
b6=1 @ 0
b5=1 @ 0
a12=1 @ 0
a11=1 @ 0
a10=1 @ 0
a9=1 @ 0
a4=1 @ 0
a3=1 @ 0
a2=1 @ 0
a1=1 @ 0
phi2=0 @ 0
```

```
Step begins @ 350 ns.
phi1=1 @ 0

Step begins @ 385 ns.
phi1=0 @ 0

Step begins @ 395 ns.
phi2=1 @ 0
state is now:
Current time= 420
clocks=0b01 cin=0 cout=0
aaaa=0b0000111100001111
bbbb=0b1111000011110000
sum=0b0000000000000000

Step begins @ 420 ns.
phi2=0 @ 0

Step begins @ 430 ns.
phi1=1 @ 0

Step begins @ 455 ns.
phi1=0 @ 0

Step begins @ 465 ns.
phi2=1 @ 0
state is now:
Current time= 490
clocks=0b01 cin=0 cout=0
aaaa=0b0000111100001111
bbbb=0b1111000011110000
sum=0b0000000000000000

Step begins @ 490 ns.
phi2=0 @ 0

Step begins @ 500 ns.
phi1=1 @ 0

Step begins @ 525 ns.
phi1=0 @ 0

Step begins @ 535 ns.
phi2=1 @ 0
s16=1 @ 14.6
s9=1 @ 16.7
s11=1 @ 16.7
s13=1 @ 16.7
s15=1 @ 16.7
s7=1 @ 16.7
s5=1 @ 16.7
s3=1 @ 16.7
s14=1 @ 16.8
s12=1 @ 16.8
s10=1 @ 16.8
```

```
s8=1 @ 16.8
s6=1 @ 16.8
s4=1 @ 16.8
s2=1 @ 17
s1=1 @ 19.1
state is now:
Current time= 560
clocks=0b01 cin=0 cout=0
aaaa=0b0000111100001111
bbbb=0b1111000011110000
sum=0b0111111111111111

Step begins @ 560 ns.
b9=1 @ 0
b1=1 @ 0
b16=0 @ 0
b15=0 @ 0
b14=0 @ 0
b13=0 @ 0
b8=0 @ 0
b7=0 @ 0
b6=0 @ 0
b5=0 @ 0
phi2=0 @ 0

Step begins @ 570 ns.
phi1=1 @ 0

Step begins @ 595 ns.
phi1=0 @ 0

Step begins @ 605 ns.
phi2=1 @ 0
state is now:
Current time= 630
clocks=0b01 cin=0 cout=0
aaaa=0b0000111100001111
bbbb=0b0000000100000001
sum=0b0111111111111111

Step begins @ 630 ns.
phi2=0 @ 0

Step begins @ 640 ns.
phi1=1 @ 0

Step begins @ 665 ns.
phi1=0 @ 0

Step begins @ 675 ns.
phi2=1 @ 0
state is now:
Current time= 700
clocks=0b01 cin=0 cout=0
aaaa=0b0000111100001111
bbbb=0b0000000100000001
```

```
sum=0b0111111111111111

Step begins @ 700 ns.
phi2=0 @ 0

Step begins @ 710 ns.
phi1=1 @ 0

Step begins @ 735 ns.
phi1=0 @ 0

Step begins @ 745 ns.
phi2=1 @ 0
s16=0 @ 14.2
s9=0 @ 16.4
s11=0 @ 16.4
s15=0 @ 16.4
s7=0 @ 16.4
s3=0 @ 16.4
s14=0 @ 16.5
s12=0 @ 16.5
s10=0 @ 16.5
s8=0 @ 16.5
s6=0 @ 16.5
s4=0 @ 16.5
s2=0 @ 16.7
s1=0 @ 20
state is now:
Current time= 770
clocks=0b01 cin=0 cout=0
aaaa=0b0000111100001111
bbbb=0b0000000100000001
sum=0b0000100000010000

Step begins @ 770 ns.
cin=1 @ 0
phi2=0 @ 0

Step begins @ 780 ns.
phi1=1 @ 0

Step begins @ 805 ns.
phi1=0 @ 0

Step begins @ 815 ns.
phi2=1 @ 0
state is now:
Current time= 840
clocks=0b01 cin=1 cout=0
aaaa=0b0000111100001111
bbbb=0b0000000100000001
sum=0b0000100000010000

Step begins @ 840 ns.
phi2=0 @ 0
```

```
Step begins @ 850 ns.
phi1=1 @ 0

Step begins @ 875 ns.
phi1=0 @ 0

Step begins @ 885 ns.
phi2=1 @ 0
state is now:
Current time= 910
clocks=0b01 cin=1 cout=0
aaaa=0b0000111100001111
bbbb=0b0000000100000001
sum=0b0000100000010000

Step begins @ 910 ns.
phi2=0 @ 0

Step begins @ 920 ns.
phi1=1 @ 0

Step begins @ 945 ns.
phi1=0 @ 0

Step begins @ 955 ns.
phi2=1 @ 0
s1=1 @ 19.1
state is now:
Current time= 980
clocks=0b01 cin=1 cout=0
aaaa=0b0000111100001111
bbbb=0b0000000100000001
sum=0b0000100000010001

Step begins @ 980 ns.
a16=1 @ 0
a15=1 @ 0
a14=1 @ 0
a13=1 @ 0
a8=1 @ 0
a7=1 @ 0
a6=1 @ 0
a5=1 @ 0
b9=0 @ 0
b1=0 @ 0
cin=0 @ 0
phi2=0 @ 0

Step begins @ 990 ns.
phi1=1 @ 0

Step begins @ 1015 ns.
phi1=0 @ 0

Step begins @ 1025 ns.
phi2=1 @ 0
```

```
state is now:
Current time= 1050
clocks=0b01 cin=0 cout=0
aaaa=0b1111111111111111
bbbb=0b0000000000000000
sum=0b0000100000010001

Step begins @ 1050 ns.
phi2=0 @ 0

Step begins @ 1060 ns.
phi1=1 @ 0

Step begins @ 1085 ns.
phi1=0 @ 0

Step begins @ 1095 ns.
phi2=1 @ 0
state is now:
Current time= 1120
clocks=0b01 cin=0 cout=0
aaaa=0b1111111111111111
bbbb=0b0000000000000000
sum=0b0000100000010001

Step begins @ 1120 ns.
phi2=0 @ 0

Step begins @ 1130 ns.
phi1=1 @ 0

Step begins @ 1155 ns.
phi1=0 @ 0

Step begins @ 1165 ns.
phi2=1 @ 0
s16=1 @ 14.6
s9=1 @ 16.7
s11=1 @ 16.7
s15=1 @ 16.7
s7=1 @ 16.7
s3=1 @ 16.7
s14=1 @ 16.8
s12=1 @ 16.8
s10=1 @ 16.8
s8=1 @ 16.8
s6=1 @ 16.8
s4=1 @ 16.8
s2=1 @ 17
state is now:
Current time= 1190
clocks=0b01 cin=0 cout=0
aaaa=0b1111111111111111
bbbb=0b0000000000000000
sum=0b0111111111111111
```

```
Step begins @ 1190 ns.
cin=1 @ 0
phi2=0 @ 0

Step begins @ 1200 ns.
phi1=1 @ 0

Step begins @ 1225 ns.
phi1=0 @ 0

Step begins @ 1235 ns.
phi2=1 @ 0
state is now:
Current time= 1260
clocks=0b01 cin=1 cout=0
aaaa=0b1111111111111111
bbbb=0b0000000000000000
sum=0b0111111111111111

Step begins @ 1260 ns.
phi2=0 @ 0

Step begins @ 1270 ns.
phi1=1 @ 0

Step begins @ 1295 ns.
phi1=0 @ 0

Step begins @ 1305 ns.
phi2=1 @ 0
state is now:
Current time= 1330
clocks=0b01 cin=1 cout=0
aaaa=0b1111111111111111
bbbb=0b0000000000000000
sum=0b0111111111111111

Step begins @ 1330 ns.
phi2=0 @ 0

Step begins @ 1340 ns.
phi1=1 @ 0

Step begins @ 1365 ns.
phi1=0 @ 0

Step begins @ 1375 ns.
phi2=1 @ 0
s16=0 @ 14.2
s9=0 @ 16.4
s11=0 @ 16.4
s13=0 @ 16.4
s15=0 @ 16.4
s7=0 @ 16.4
s5=0 @ 16.4
s3=0 @ 16.4
```

```
s14=0 @ 16.5
s12=0 @ 16.5
s10=0 @ 16.5
s8=0 @ 16.5
s6=0 @ 16.5
s4=0 @ 16.5
s2=0 @ 16.7
s1=0 @ 20
cout=1 @ 21.1
state is now:
Current time= 1400
clocks=0b01 cin=1 cout=1
aaaa=0b1111111111111111
bbbb=0b0000000000000000
sum=0b1000000000000000

Step begins @ 1400 ns.
h1=1 @ 0
cin=0 @ 0
ph12=0 @ 0

Step begins @ 1410 ns.
ph11=1 @ 0

Step begins @ 1435 ns.
ph11=0 @ 0

Step begins @ 1445 ns.
ph12=1 @ 0
state is now:
Current time= 1470
clocks=0b01 cin=0 cout=1
aaaa=0b1111111111111111
bbbb=0b0000000000000001
sum=0b1000000000000000

Step begins @ 1470 ns.
ph12=0 @ 0

Step begins @ 1480 ns.
ph11=1 @ 0

Step begins @ 1505 ns.
ph11=0 @ 0

Step begins @ 1515 ns.
ph12=1 @ 0
state is now:
Current time= 1540
clocks=0b01 cin=0 cout=1
aaaa=0b1111111111111111
bbbb=0b0000000000000001
sum=0b1000000000000000

Step begins @ 1540 ns.
ph12=0 @ 0
```

```
Step begins @ 1550 ns.
phi1=1 @ 0

Step begins @ 1575 ns.
phi1=0 @ 0

Step begins @ 1585 ns.
phi2=1 @ 0
state is now:
Current time= 1610
clocks=0b01 cin=0 cout=1
aaaa=0b1111111111111111
bbbb=0b0000000000000001
sum=0b1000000000000000

Step begins @ 1610 ns.
b1=0 @ 0
phi2=0 @ 0

Step begins @ 1620 ns.
phi1=1 @ 0

Step begins @ 1645 ns.
phi1=0 @ 0

Step begins @ 1655 ns.
phi2=1 @ 0
state is now:
Current time= 1680
clocks=0b01 cin=0 cout=1
aaaa=0b1111111111111111
bbbb=0b0000000000000000
sum=0b1000000000000000

Step begins @ 1690 ns.
a16=0 @ 0
a15=0 @ 0
a14=0 @ 0
a13=0 @ 0
a12=0 @ 0
a11=0 @ 0
a10=0 @ 0
a9=0 @ 0
a8=0 @ 0
a7=0 @ 0
a6=0 @ 0
a5=0 @ 0
a4=0 @ 0
a3=0 @ 0
a2=0 @ 0
a1=0 @ 0
phi2=0 @ 0

Step begins @ 1690 ns.
phi1=1 @ 0
```

```
Step begins @ 1715 ns.
phi1=0 @ 0

Step begins @ 1725 ns.
phi2=1 @ 0
state is now:
Current time= 1750
clocks=0b01 cin=0 cout=1
aaaa=0b0000000000000000
bbbb=0b0000000000000000
sum=0b1000000000000000

Step begins @ 1750 ns.
b16=1 @ 0
b15=1 @ 0
b14=1 @ 0
b13=1 @ 0
b12=1 @ 0
b11=1 @ 0
b10=1 @ 0
b9=1 @ 0
b8=1 @ 0
b7=1 @ 0
b6=1 @ 0
b5=1 @ 0
b4=1 @ 0
b3=1 @ 0
b2=1 @ 0
b1=1 @ 0
phi2=0 @ 0

Step begins @ 1760 ns.
phi1=1 @ 0

Step begins @ 1785 ns.
phi1=0 @ 0

Step begins @ 1795 ns.
phi2=1 @ 0
s16=1 @ 14.6
s9=1 @ 16.7
s11=1 @ 16.7
s13=1 @ 16.7
s15=1 @ 16.7
s7=1 @ 16.7
s5=1 @ 16.7
s3=1 @ 16.7
s14=1 @ 16.8
s12=1 @ 16.8
s10=1 @ 16.8
s8=1 @ 16.8
s6=1 @ 16.8
s4=1 @ 16.8
s2=1 @ 17
s1=1 @ 19.1
```

```
cout=0 @ 22.9
state is now:
Current time= 1820
clocks=0b01 cin=0 cout=0
aaaa=0b0000000000000000
bbbb=0b1111111111111111
sum=0b0111111111111111

Step begins @ 1820 ns.
a12=1 @ 0
ph12=0 @ 0

Step begins @ 1830 ns.
ph11=1 @ 0

Step begins @ 1855 ns.
ph11=0 @ 0

Step begins @ 1865 ns.
ph12=1 @ 0
s16=0 @ 14.2
s9=0 @ 16.4
s11=0 @ 16.4
s13=0 @ 16.4
s15=0 @ 16.4
s7=0 @ 16.4
s5=0 @ 16.4
s3=0 @ 16.4
s14=0 @ 16.5
s12=0 @ 16.5
s10=0 @ 16.5
s8=0 @ 16.5
s6=0 @ 16.5
s4=0 @ 16.5
s2=0 @ 16.7
s1=0 @ 20
state is now:
Current time= 1890
clocks=0b01 cin=0 cout=0
aaaa=0b0000100000000000
bbbb=0b1111111111111111
sum=0b0000000000000000

Step begins @ 1890 ns.
b12=0 @ 0
ph12=0 @ 0

Step begins @ 1900 ns.
ph11=1 @ 0

Step begins @ 1925 ns.
ph11=0 @ 0

Step begins @ 1935 ns.
ph12=1 @ 0
s16=1 @ 14.6
```

```
s9=1 @ 16.7
s11=1 @ 16.7
s13=1 @ 16.7
s15=1 @ 16.7
s7=1 @ 16.7
s5=1 @ 16.7
s3=1 @ 16.7
s14=1 @ 16.8
s12=1 @ 16.8
s10=1 @ 16.8
s8=1 @ 16.8
s6=1 @ 16.8
s4=1 @ 16.8
s2=1 @ 17
s1=1 @ 19.1
state is now:
Current time= 1960
clocks=0b01 cin=0 cout=0
aaaa=0b0000100000000000
bbbb=0b1111011111111111
sum=0b0111111111111111

Step begins @ 1960 ns.
cin=1 @ 0
phi2=0 @ 0

Step begins @ 1970 ns.
phi1=1 @ 0

Step begins @ 1995 ns.
phi1=0 @ 0

Step begins @ 2005 ns.
phi2=1 @ 0
s16=0 @ 14.2
s13=0 @ 16.4
s15=0 @ 16.4
s14=0 @ 16.5
s12=0 @ 16.5
cout=1 @ 21.1
state is now:
Current time= 2030
clocks=0b01 cin=1 cout=1
aaaa=0b0000100000000000
bbbb=0b1111011111111111
sum=0b1000011111111111

Step begins @ 2030 ns.
b16=0 @ 0
b15=0 @ 0
b14=0 @ 0
b13=0 @ 0
b11=0 @ 0
b10=0 @ 0
b9=0 @ 0
b8=0 @ 0
```

```
b7=0 @ 0
b6=0 @ 0
b5=0 @ 0
b4=0 @ 0
b3=0 @ 0
b2=0 @ 0
b1=0 @ 0
a12=0 @ 0
ph12=0 @ 0

Step begins @ 2040 ns.
ph11=1 @ 0

Step begins @ 2065 ns.
ph11=0 @ 0

Step begins @ 2075 ns.
ph12=1 @ 0
s16=1 @ 14.6
s13=1 @ 16.7
s15=1 @ 16.7
s14=1 @ 16.8
s12=1 @ 16.8
cout=0 @ 22.9
state is now:
Current time= 2100
clocks=0b01 cin=1 cout=0
aaaa=0b0000000000000000
bbbb=0b0000000000000000
sum=0b0111111111111111

Step begins @ 2100 ns.
cin=0 @ 0
ph12=0 @ 0

Step begins @ 2110 ns.
ph11=1 @ 0

Step begins @ 2135 ns.
ph11=0 @ 0

Step begins @ 2145 ns.
ph12=1 @ 0
s16=0 @ 14.2
s9=0 @ 16.4
s11=0 @ 16.4
s13=0 @ 16.4
s15=0 @ 16.4
s7=0 @ 16.4
s5=0 @ 16.4
s3=0 @ 16.4
s14=0 @ 16.5
s12=0 @ 16.5
s10=0 @ 16.5
s8=0 @ 16.5
s6=0 @ 16.5
```

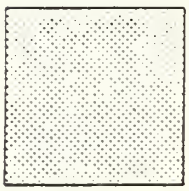
```
s4=0 @ 16.5
s2=0 @ 16.7
s1=0 @ 20
cout=1 @ 21.1
state is now:
Current time= 2170
clocks=0b01 cin=0 cout=1
aaaa=0b0000000000000000
bbbb=0b0000000000000000
sum=0b1000000000000000

Step begins @ 2170 ns.
phi2=0 @ 0

Step begins @ 2180 ns.
phi1=1 @ 0

Step begins @ 2205 ns.
phi1=0 @ 0

Step begins @ 2215 ns.
phi2=1 @ 0
cout=0 @ 22.9
state is now:
Current time= 2240
clocks=0b01 cin=0 cout=0
aaaa=0b0000000000000000
bbbb=0b0000000000000000
sum=0b0000000000000000

Step begins @ 2240 ns.
phi2=0 @ 0

Step begins @ 2250 ns.
phi1=1 @ 0

Step begins @ 2275 ns.
phi1=0 @ 0

Step begins @ 2285 ns.
phi2=1 @ 0
s1=0 @ 20
state is now:
Current time= 2310
clocks=0b01 cin=0 cout=0
aaaa=0b0000000000000000
bbbb=0b0000000000000000
sum=0b0000000000000000

Step begins @ 2310 ns.
phi2=0 @ 0

Step begins @ 2320 ns.
phi1=1 @ 0

Step begins @ 2345 ns.
```

```
phi1=0 @ 0

Step begins @ 2355 ns.
phi2=1 @ 0
state is now:
Current time= 2380
clocks=0b01 cin=0 cout=0
aaaa=0b0000000000000000
bbbb=0b0000000000000000
sum=0b0000000000000000


exit
```

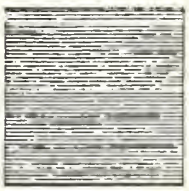
APPENDIX E
LAYOUTS

LEGEND

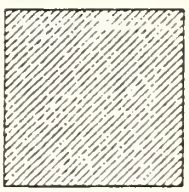
Contact Cut



p-well

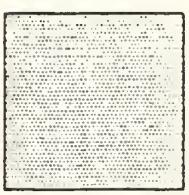


P+ doping

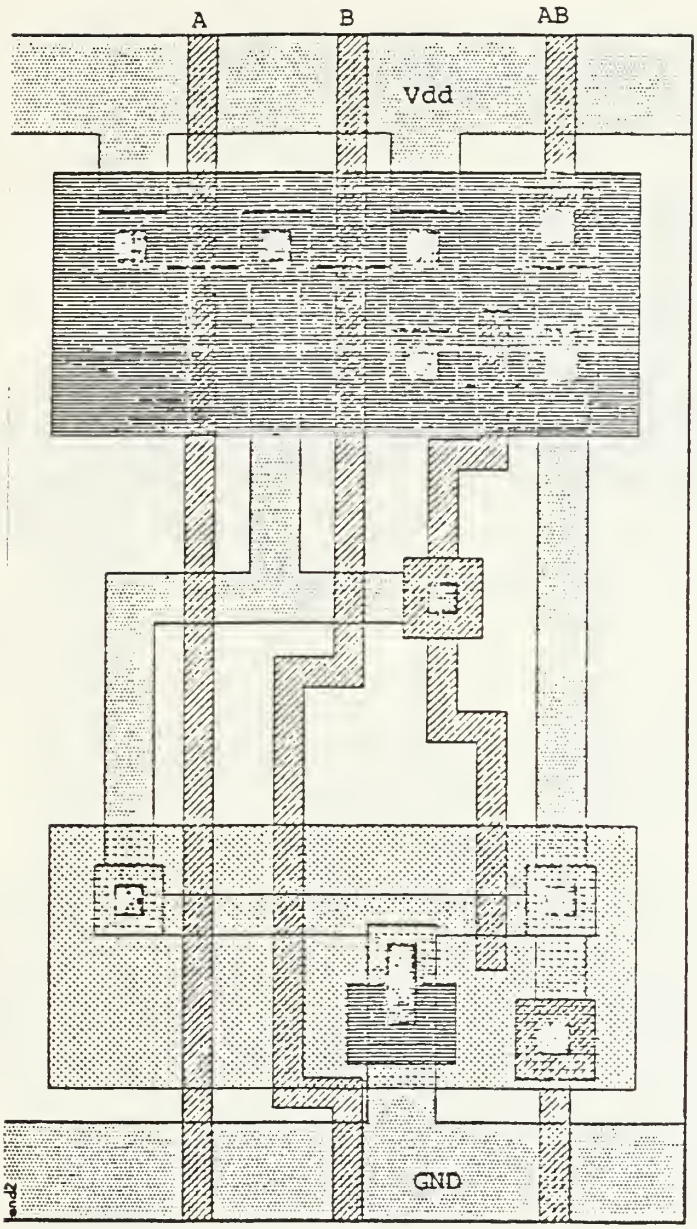


polysilicon

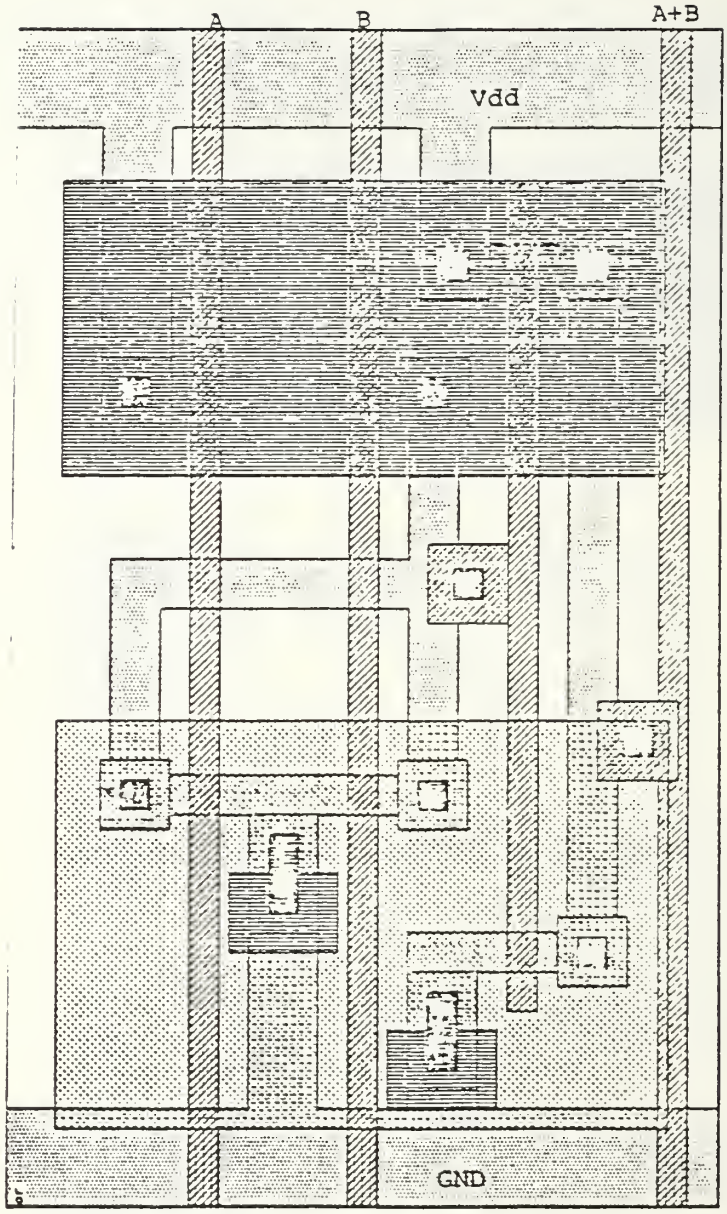
Diffusion



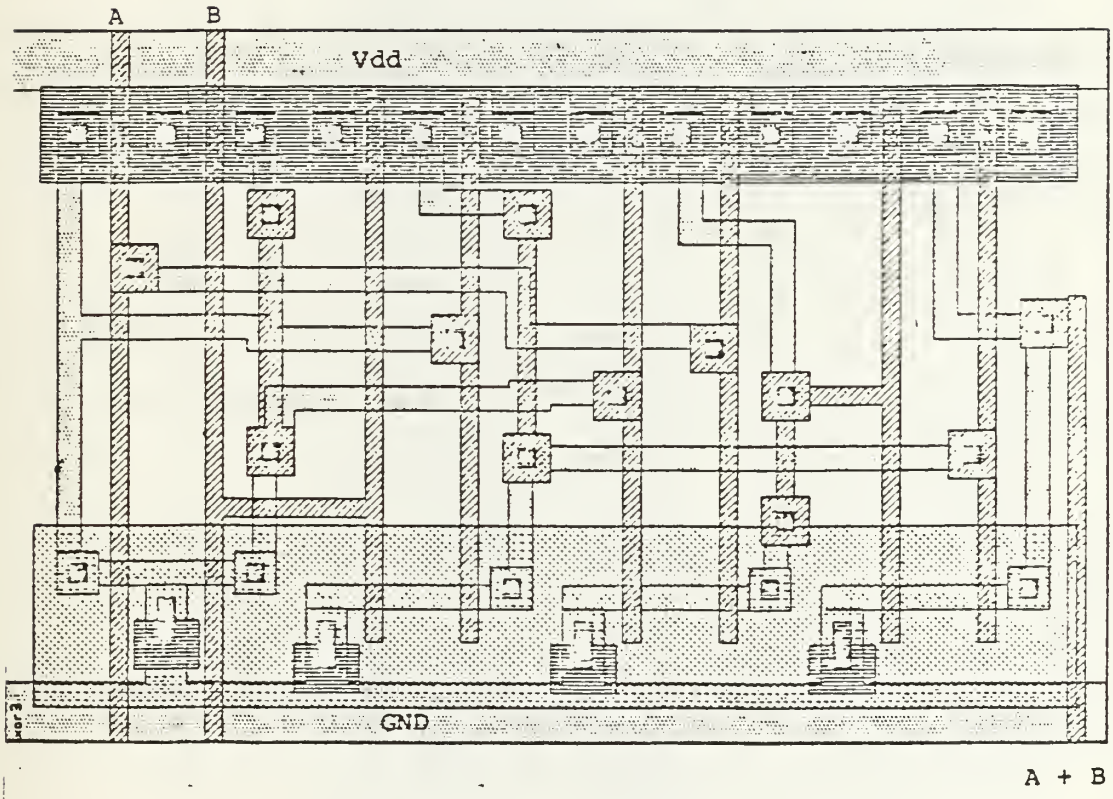
Metal



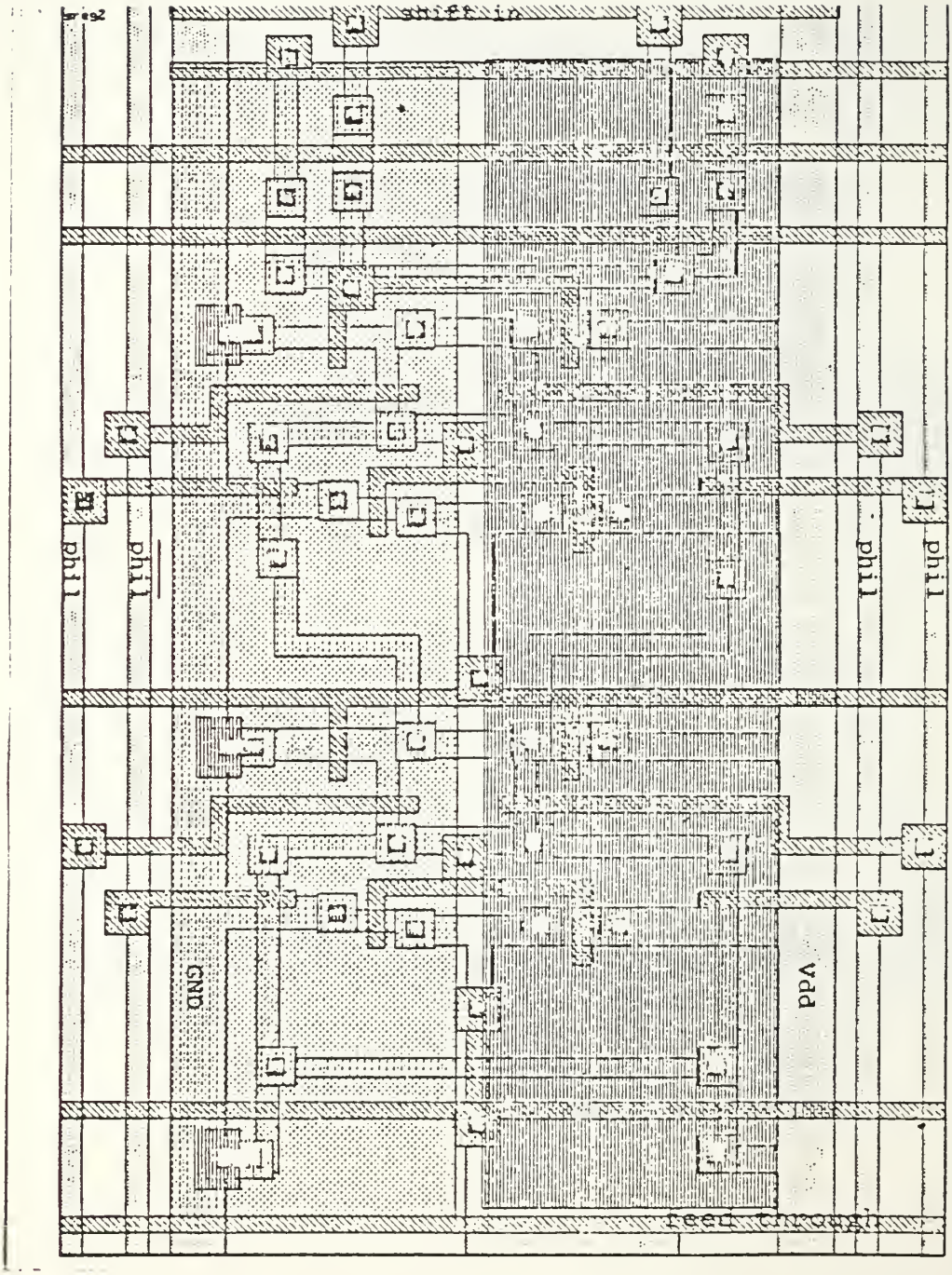
AND Gate

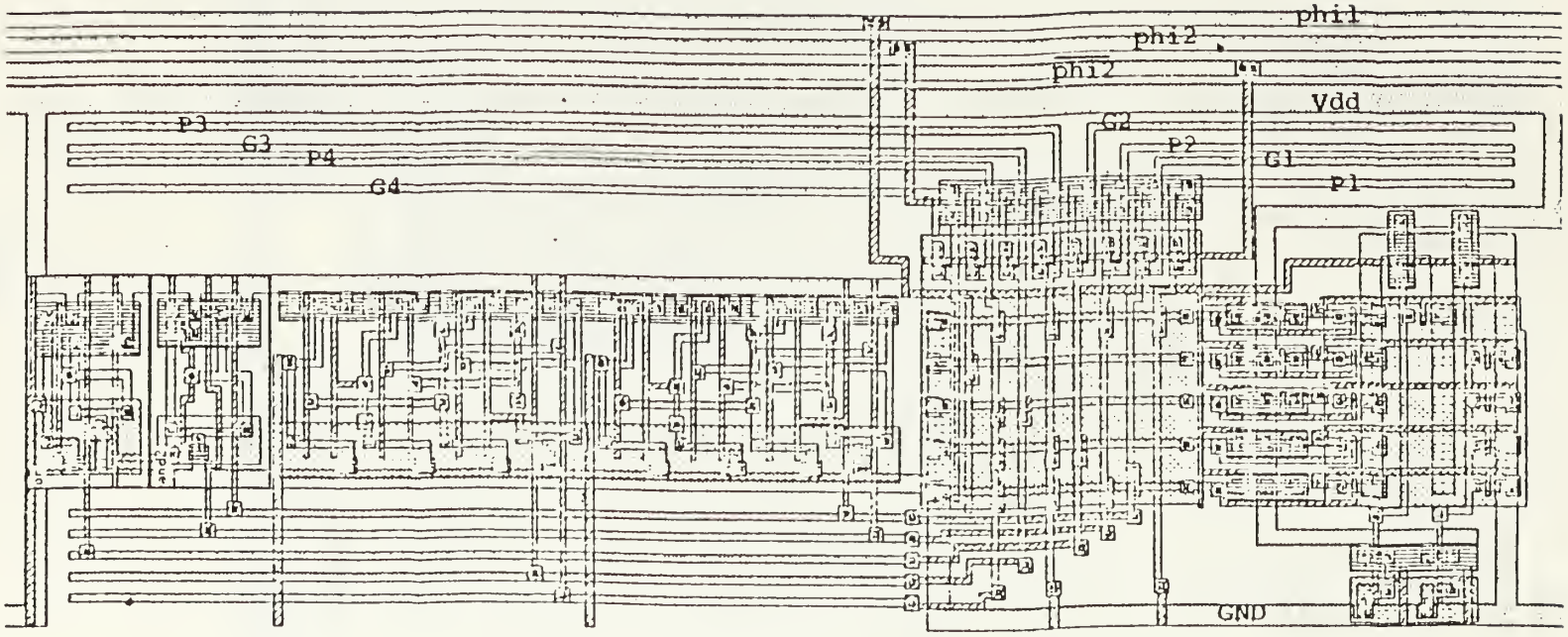


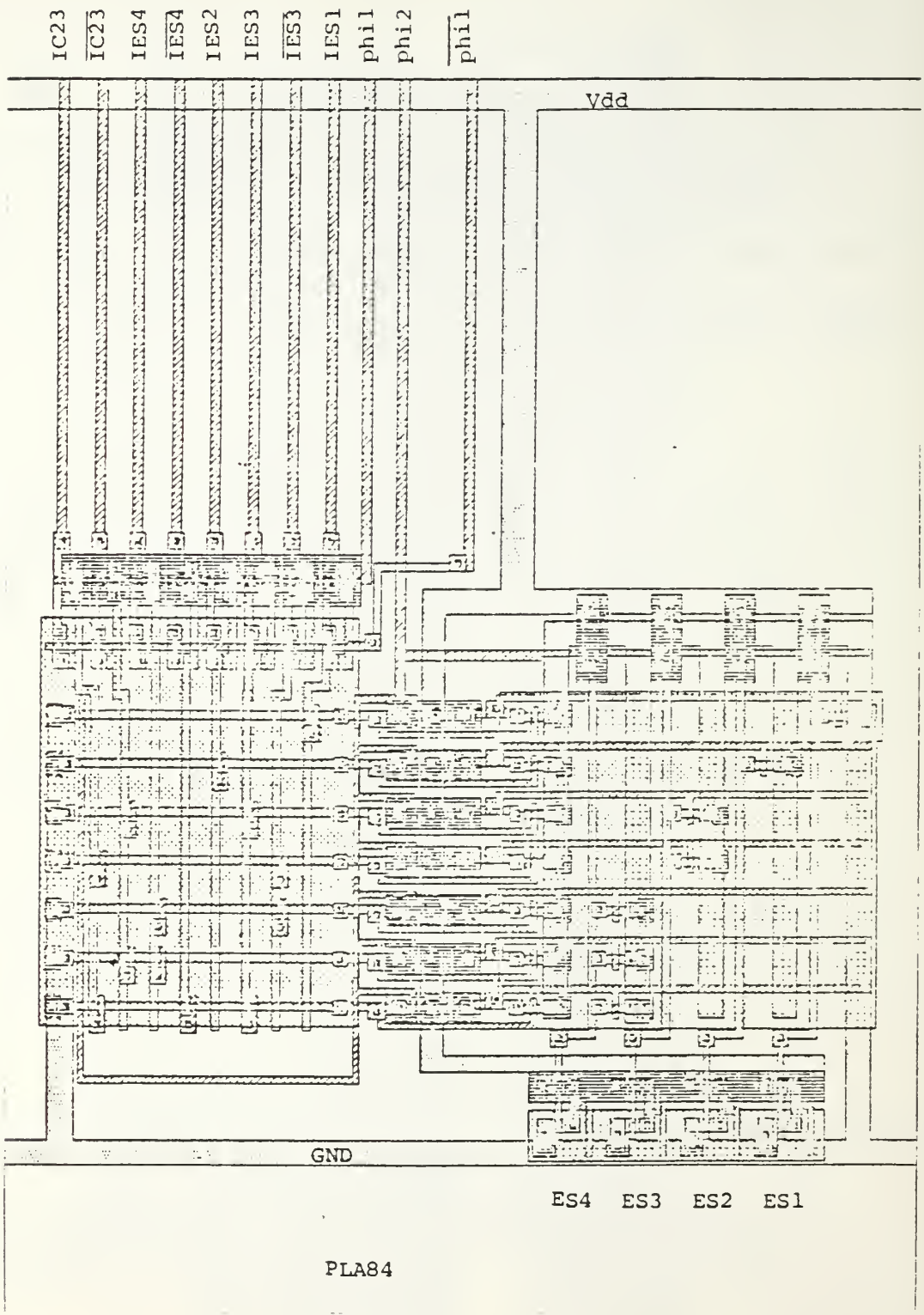
OR Gate

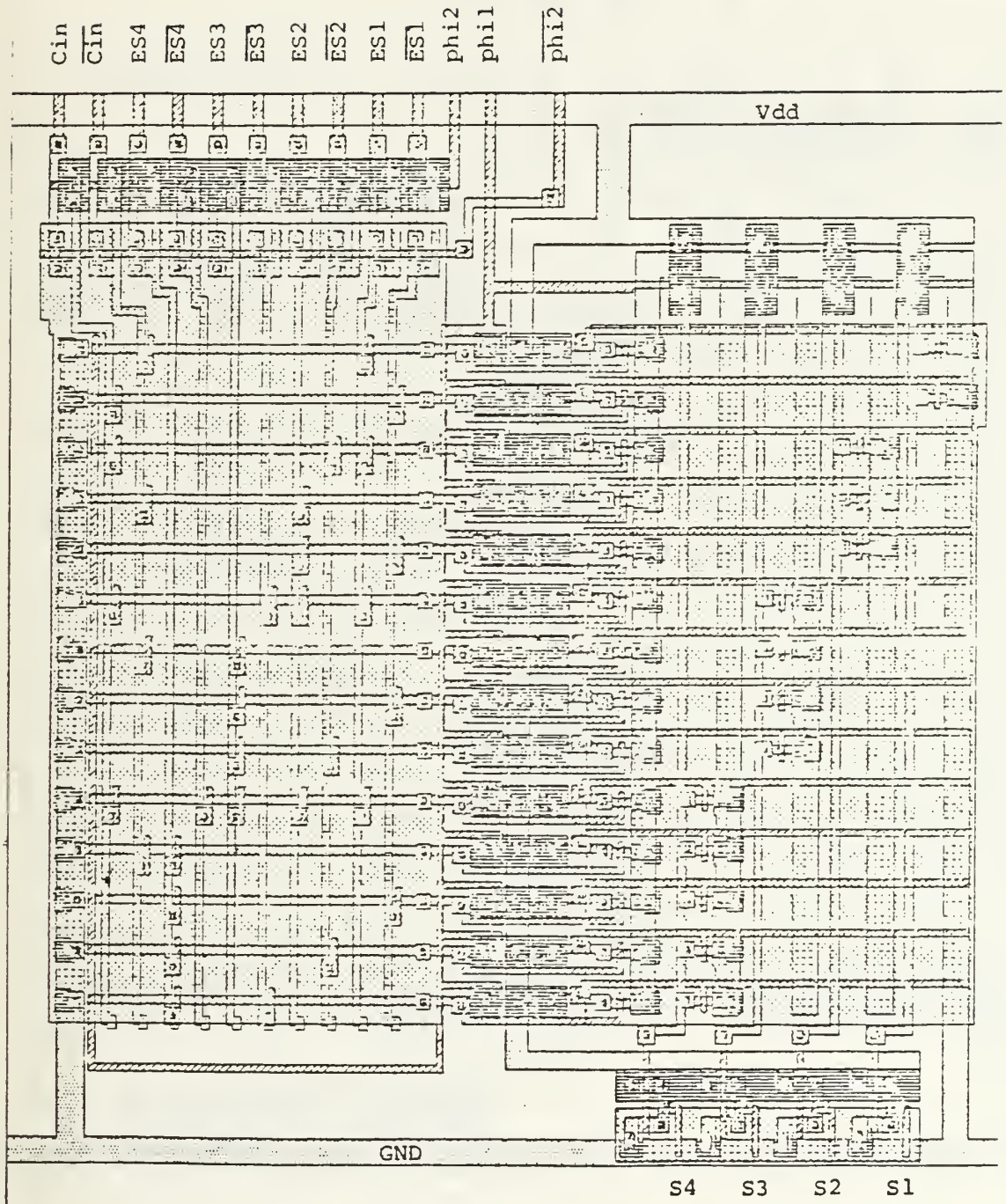


XOR Gate

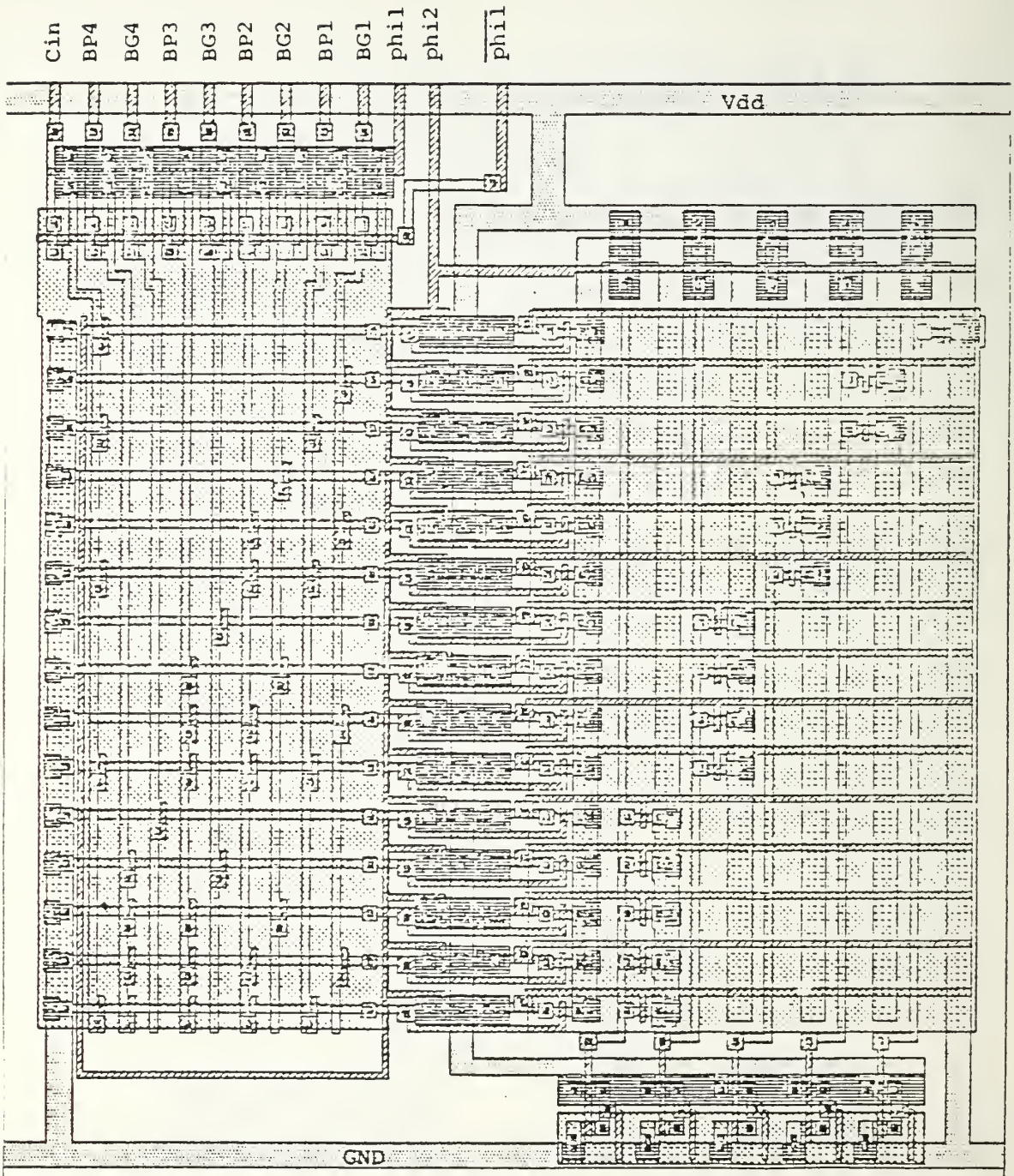

shift in
in
CON(n)
$\overline{CON(n)}$
phi1
$\overline{phi1}$
phi1
$\overline{phi1}$
out
GND
Vdd
shift out
feed through


phi1
phi2
phi2
Vdd
P3
G3
P4
G4
G2
P2
G1
P1
GND
IC23
IES4
IES2
PLA82
IES3
IES1
BP
BG


IC23
$\overline{IC23}$
IES4
$\overline{IES4}$
IES2
IES3
$\overline{IES3}$
IES1
phi1
phi2
$\overline{phi1}$
Vdd
GND
ES4 ES3 ES2 ES1
PLA84


Cin
$\overline{Cin}$
ES4
$\overline{ES4}$
ES3
$\overline{ES3}$
ES2
$\overline{ES2}$
ES1
$\overline{ES1}$
phi2
phi1
$\overline{phi2}$
Vdd
GND
S4
S3
S2
S1

PLA104


Cin
BP4
BG4
BP3
BG3
BP2
BG2
BP1
BG1
phi1
phi2
$\overline{phi1}$
Vdd
GND
BC4
BC3
BC2
BC1
BC0

PLA915

APPENDIX F
TEST VECTORS

Addend A	Addend B	Cin	Sum
msb- - - - - lsb	msb- - - - - lsb		msb- - - - - -lsb
initialize all internal nodes			
0000000000000000	0000000000000000	0	XXXXXXXXXXXXXXXX
0000000000000000	0000000000000000	0	XXXXXXXXXXXXXXXX
0000000000000000	0000000000000000	0	0000000000000000
test for proper P and G primitives			
0000000000000000	1111111111111111	0	0111111111111111
1111111111111111	0000000000000000	0	0111111111111111
0101010101010101	1010101010101010	0	0111111111111111
1010101010101010	0101010101010101	0	0111111111111111
test fcr proper IES			
0001000100010001	0000000000000000	0	0001000100010001
0001000100010001	0001000100010001	0	0001000100010001
0101010101010101	0001000100010001	0	0011001100110011
0101010101010101	0101010101010101	0	0101010101010101
test fcr proper IC23			
0101010101010101	0011001100110011	0	0100010001000100
0010001000100010	0011001100110011	0	0010101010101010
test for carry from block to blcck			
0000000000001111	0000000000000001	0	0000000000010000
0000000000001111	0000000000000000	1	0000000000010000
0000000011111111	0000000000000000	1	0000000100000000

0000000011111111	0000000000000001	0	0000000010000000
0000000011111111	0000000000010000	0	0000000010000111
0000111111111111	0000000000000000	1	0000100000000000
0000111111111111	0000000000000001	0	0000100000000000
0000111111111111	0000000000010000	0	0000100000000111
0000111111111111	0000000010000000	0	0000100000011111
1111111111111111	0000000000000000	1	1000000000000000
1111111111111111	0000000000000001	0	1000000000000000
1111111111111111	0000000000010000	0	1000000000000111
1111111111111111	0000000010000000	0	1000000000011111
1111111111111111	0001000000000000	0	1000011111111111

LIST OF REFERENCES

1. Mead, C. and Conway, L., Introduction to VLSI Systems, Addison-Wesley, 1980.

2. Carlson, D.J., Application of a Silicon Compiler to VLSI Design of Digital Pipelined Multipliers, MSEE Thesis, Naval Postgraduate School, Monterey, Ca., June 1984.

3. Conradi, J. R. and Hauenstein, B. R., VLSI Design of a 16 Bit Very Fast Pipelined Carry Look Ahead Adder, MSEE Thesis, Naval Postgraduate School, Monterey, Ca., September 1983.

4. Ousterhout, J., Editing VLSI Circuits with Caesar, Computer Science Division, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, pp. 1-22, March 22,1983.

5. Tsai, L. L. and Achugbue, J. O., "BURLAP: A Hierarchical VLSI Design System," VLSI Design, pp. 21-26, July/August 1983.

6. Carnegie-Mellon University Computer Science Department Report CMU-CS-84-101, Let's Design CMOS Circuits! Part One, by M. Annaratone, April 3, 1984.

7. Krambeck, R. H., Lee, C. M. and Law, H. S., "High Speed Compact Circuits with CMOS," IEEE Journal of Solid State Circuits, Vol. SC-17, No. 3, pp. 614-619, June, 1982.

8. Fang, R. C. and Moll, J. L., "Latchup Model for the Parasitic p-n-p-n Path in Bulk CMOS," IEEE Transactions on Electron Devices, Vol. ED-31, No. 1, pp. 113-120, January 1984.

9. Massachusetts Institute of Technology VLSI Memo No. 82-117, Introductory CMOS Techniques, by L. A. Glasser and W. S. Song, February 1983.

10. Computer Science Division (EECS), University of California, Berkeley, Report No. UCB/CSD/83/115, 1983 VLSI Tools, edited by R. N. Mayo, J. K. Ousterhout, and W. S. Scott, March, 1983.

11. University of Washington/Northwest VLSI Consortium, VLSI Design Tools Reference Manual, Release 2.0, August 1, 1984.

12. Flores, I., The Logic of Computer Arithmetic, Prentice-Hall, 1963.

13. Williams, T. W., "Design for Testability: What's the Motivation?" VLSI Design, October, 1983.

BIBLIOGRAPHY

Mercer, M. R. and Agarwal, V. D., "A Novel Clocking Technique for VLSI Circuit Testability," IEEE Journal of Solid State Circuits, Vol. SC-19, No. 2, pp. 207-211, April, 1984.

Tosuntikool, N. and Saxe, C. L., "Rapid Design of Functional Cells," VLSI Design, pp. 73-77, July/August 1983.

Williams, M. J. Y. and Angell, J. B., "Enhancing Testability of Large-Scale Integrated Circuits via Test Points and Added Logic," IEEE Transactions on Computers, Vol. C-22, No. 1, pp. 46-60, January, 1973.

INITIAL DISTRIEUTION LIST

	No. Copies
1. Superintendent Attn: Library, Code 0142 Naval Postgraduate School Monterey, California 93943	2
2. Dr. Dcnald Kirk Code 62KI Naval Postgraduate School Monterey, California 93943	6
3. Dr. H. H. Loomis Code 62LM Naval Postgraduate School Monterey, California 93943	1
4. Dr. M. L. Cotton Code 62CC Naval Postgraduate School Monterey, California 93943	1
5. Defense Technical Informatior Center Cameron Station Alexandria, Virginia 22314	2
6. LCDR William R. Reid 11224 Edgemoor Court Woodbridge, Virginia 22192	1

Thesis
R3255
c.1

Reid
Design of a sixteen
bit pipelined adder
using CMOS Bulk P-Well
Technology.